LINUX
파일체계
룡남산
BLOCK GROUP
BLOCK GROUP
BLOCK GROUP
super block
group descriptor
block bitmap
inode bitmap
data block 0
data block1
data block n-1
data block n
김일성종합대학출판사
주체91

Linux파일체계

김일성종합대학출판사

차 례

제1장. 개 괄

제2장. 핵심부의 콤파일

제3장. 파일체계란 무엇인가

제4장. 가상파일체계 VFS

제5장. 론리기록권관리기 LVM

제6장. Linux용 RAID

제7장. 2차확장파일체계

제８장. Linux용실행기록 파일체계

제９장. Linux용Reiser 파일체계

제１０장. 확장파일체계

부록1. 쏘프트웨어 RAID의 리용방법

부록2. 참고문헌 336

부록3. Loopback뿌리파일체계의 리용방법

부록4. Linux의 구획설정방법

제1장. 개 괄

UNIX(유닉스)에 대하여 오래동안 연구도 해왔고 일정한 기간 Linux(리눅스)를 사용해 온 연구자들까지도 일반적으로 Linux에서 자료를 어떻게 기억시키며 어떻게 검색하는가에 대해서는 잘 알고 있다고 생각해 왔다.

그런데 지난 2000년 가을에 있은 UNIX토론회에서는 정규 Linux파일체계*[1]로부터 일단 잃어 졌다고 생각되는 파일을 아주 쉽게 다시 회복시켰다는 내용이 제기되여 열린 파일이 지워 지면 그 파일을 다시 회복시킬수 없다고 한 주장이 사라지게 되였다.

이로부터 Linux에서 체계의 믿음성과 안정성보호문제가 중요하게 제기되였다.

GNU/Linux와 파일체계

최초에 Linux는 Minix(미닉스)*[2]조작체계하에서 교차개발되였다. 리누스 토발즈(Linus Torvalds)는 처음부터 새로운 파일체계를 설계하는것보다는 두개의 체계사이에서 디스크를 공유할수 있다는데로부터 이러한 방법을 선택하였다.

Minix파일체계는 아주 효과적이였으며 상대적으로 오유에 안정한 쏘프트웨어였다. 그러나 Minix파일체계는 제한조건이 너무 강하였기때문에 사람들은 점차 Linux에서 새로운 파일체계를 실현하기 위한 연구를 시작하게 되였다.

Linux핵심부(Linux kernel)에 새로운 파일체계를 보다 쉽게 첨가하기 위하여 가상파일체계층(Virtual File System Layer)이 개발(Chris Provenzano에 의하여)되였는데 이 가상파일체계층은 후에 Linux핵심부로 종합완성되기에 앞서 리누스 토발즈에 의하여 다시 서술되였다(이 내용에 대해서는 이 책의 가상파일체계에서 설명한다.). 가상파일체계가 Linux의 핵심부로 완성된후 확장파일체계(extended File System (ext))라고 부르는 새로운 파일체계가 1992년 4월에 실현되였으며 Linux 판본 0.96으로 보충되였다.

이 새로운 파일체계는 Minix의 두가지 큰 제한성을 극복하였는데 하나는 최대파일크기가 2GB인것이고 다른 하나는 최대파일이름길이가 255문자인것이였다.

ext파일체계는 Minix파일체계에 비해서는 훨씬 개선되였으나 여전히 자체의 부족점을 가지고 있었다.

그러한 부족점들로는 접근분리, 색인마디변경, 자료변경시간표식*[3] 등을 지원하지 못하는것을 들수 있다.

---

*[1] Linux ext2파일체계에서 어떤 파일을 지울 때 등록부입구점은 지워 지지 않는다. 결과 적어도 등록부입구점이 재쓰기되기전까지는 이름대색인마디넘기기가 보존된다. 그러므로 사용자는 등록부행블로크들에 접근하여 지워 진 파일의 이름과 색인마디번호를 찾을수 있으며 따라서 첫 12개 혹은 직접 블로크에 접근할수 있다.

*[2] Minix와 그 파일체계는 교육목적으로 개발되였다(Andrew Taneu baum에 의하여).

*[3] 이 특성은 다음장들에서 설명한다.

ext파일체계는 그 파일체계안의 자유블로크와 색인마디들을 쉽게 추적하기 위하여 핵심부안에서 련결목록을 리용하였으나 역시 새로운 불합리성에 직면하였다. 즉 파일체계리용은 개선되였으나 련결목록이 정렬되지 않고 파일체계가 토막으로 분리되는것으로 하여 불합리한 파일체계의 겹침현상이 초래되게 되였다.

이러한 문제점들에 대응하여 두개의 새로운 파일체계가 1993년초에 발표되였는데 하나는 Xia파일체계(Xiafs)이고 다른 하나는 2차확장파일체계(Second Extended File System) 즉 ext2fs였다.

Xia파일체계는 기본적으로 Minix파일체계에 기초하였으며 다만 몇가지 기본개선대책만을 보충하였다.

이 파일체계는 Minix처럼 긴 파일이름을 제공하며 보다 큰 구획(Partition)도 지원하며 세가지 시간표식기능 즉 시간생성, 시간변경, 시간접근기능을 지원한다.

한편 ext2fs는 확장파일체계코드에 기초하였으며 상당히 개선되였다.

이 파일체계는 그 시작으로부터 보다 개선된 성능확장에 목적을 두고 설계되였다. 두개의 새로운 파일체계가 처음으로 발표되였을 때 그것들은 서로 다른 실현방식과 원천코드를 가지고 있지만 본질적으로는 같은 특성을 제공하였다.

또한 설계의 최소화에 의하여 Xiafs는 실제적으로 ext2fs보다 더 안정하였다. 결국 ext2fs에서는 오유가 고정되고 많이 개선되여 새로운 특성들이 보충되였다.

오늘 ext2fs는 대단히 안정하며 사실에 있어서 표준 Linux파일체계로 되였다.

이 책은 ext2fs의 넓은 범위와 함께 현재 Linux에서 리용가능한 모든 중요한 파일체계들에 대하여 고찰하며 그것들의 우점과 결점을 시험하고 어떻게 효과적으로 리용하겠는가를 고찰한다.

이 책의 목적

이 책은 현재 Linux용으로 리용할수 있는 가장 중요한 파일체계들에 대한 일반적리해를 더 깊이 할 목적으로 서술되였다.

파일체계를 완전히 리해하려면 그 파일체계가 어떻게 서술되였는가를 때때로 알아 보아야 한다.

그러나 이 파일체계들의 원천코드를 해설하는것이 아니라 언제 어느 파일체계를 효과적으로 사용하겠는가를 보여 주는것이 이 책의 목적이다. 아래에 광범히 쓰이는 몇개의 용어들을 소개한다.

▼ 핵심부(kernel)는 보호방식에서 실행되며 장치의 특권준위등록기들에 접근할수 있는 조작체계프로그람이다.

- 파일체계는 사용자나 체계자원의 서로 독립적인 저장통을 표현하는 조종블로크들의 론리적집합이다. 《파일체계》라는 용어를 쓸 때는 일정한 애매성이 존재한다. 용어를 리용하는 한가지 측면은 ext2fs나 NFS와 같은 특정한 파일체계의 종류에 대하여 고찰할 때이며 다른 한 측면은 /usr나 혹은 /boot와 같은 파일체계의 특정한 실례를 고찰할 때이다.

- 이름공간(name space)은 파일이름과 같이 일의적으로 지정된 식별자들의 모임이다. 이름공간안에서 파일이름은 한가지 실례에 불과하다. 일반적으로 이름공간은 등록부범위안에 포함된다.
- ▲ 등록부(directory)는 파일체계에 의하여 유지관리되는 특별한 파일이다. 등록부에는 입구점(entry)들의 목록이 포함된다. 사용자에게 있어서 어떤 입구점은 파일이름으로 표현되며 그것은 기호로 표현된 입구점이름에 의하여 접근되는데 이때의 기호입구점이름은 사용자의 파일이름으로 된다.

이 책을 읽어야 할 사람

이 책은 체계관리자, 망관리자, 개발자의 능력을 높여 주자는데 그 목적이 있다. 그러므로 이 책은 또한 장치와 프로그람에 대한 일반적인 리해를 더 잘 가지려고 하는 Linux애호가들에게도 호감을 가지게 한다. 다음장들에서 체계관리자들은 특정한 파일체계에 핵심부를 어떻게 준비하며 어느 체계를 리용할수 있으며 그것들을 어떻게 정확하게 사용하겠는가를 배우게 된다. 또한 자연스러운 환경에서 파일체계를 전환하여 상당한 정도로 체계성능을 높여 나갈수 있게 될것이다. 개발자들은 자기들의 응용실천에 파일체계가 어떻게 영향을 주는가를 알게 된다. 많은 사람들이 개발자들에게는 파일체계가 실지로 명백하다고 말하고 있지만 사실은 그렇지 않다. 실례로 잠금기술이 파일체계에 의하여 효과적으로 실현된다는것을 안다는것은 프로그람작성자나 체계개발자들로 하여금 이 기능성을 응용한 코드를 만들지 않아도 되게 한다.

이 책을 읽기전에 알아야 할 내용

콤퓨터과학리론의 일반적개념 특히 이름공간구역이나 입출력개념을 잘 가지는것은 아주 중요하다.

또한 독자가 Linux대면부의 동작지식과 기본체계관리에 무엇이 포함되는가에 대한 기초적인 리해를 가지고 있다면 더욱 좋다.

파일체계에 대한 예비지식은 없어도 된다. 제공되고 있는 대다수의 코드들을 알고 있다고 해도 C언어프로그람을 읽을수 있는 편이 더 좋다.

C언어에 대하여서는 C언어의 설계자들인 Kernighan/Ritche가 쓴 C Programming Language를 소개할수 있다.

이 책의 내용

이 책은 Linux조작체계에 대한 간단한 요약과 함께 그것을 조작하는 방법을 내용으로 하고 있다. 또한 책은 Linux핵심부의 재콤파일을 어떻게 진행하는가에 대하여 서술하고 있는바 이것은 표준Linux배포판핵심부로 선행콤파일되지 않은 파일체계의 리용에서 중요한 지식으로 된다. 일반 UNIX의 파일체계에 대하여 고찰해 보면 가상파일체계(Virtual File system : VFS)를 통하여 Linux의 파일체계가상화를 알게 되며 따라서 모든 중요한 파일체계들을 고찰하고 설명할수 있다.

이 책에서는 리용가능한 모든 Linux파일체계에 대하여 일일이 설명하지 않고 가장 중요한것들과 그것들을 어떻게 효과적으로 리용하겠는가에 대해서만 관심을 돌려 취급한다.

이 책의 리용방법

이 책은 글줄을 따라서 처음부터 마지막까지 읽는것이 제일 좋다. 처음에 한번 읽어본후에는 매일매일 파일체계를 리용하면서 속성참고서로 리용해도 좋다.

보다 자세한 내용알아보기

Linux파일체계에 대한 가장 최신자료는 인터네트상에서 찾아 볼수 있지만 그것은 사실상 행으로 된 자료에 불과하며 이 자료를 수집하여 그것을 조립하고 론리정연한 형식으로 정리한다는것은 쉬운 일이 아니다. Linux파일체계의 가장 큰 가치는 Web싸이트, 원천코드, 론문 등을 통하여 공개적으로 리용가능한 자료로 론리정연하게 잘 연구되였으며 형식화된 원천들을 제공한다는데 있다.

핵심부상에서 훌륭한 정보원천의 하나는 물론 파일체계를 실현하는 핵심부원천코드이다.

이 책에서 언급한 모든 파일체계에 대한 개발자의 우편목록에 대한 서명은 파일체계에 관계되는 모든 측면을 알아 내는데서 아주 중요한 방법의 하나이다.

의견과 해설

임의의 의견이나 해설에 대해서는 언제나 접수하며 moshe_bar@hotmail.com이라든지 혹은 Moshe Bar c/o McGraw-Hill, professional Book Group, Two Penn Plaza, New York, NY 10121주소를 리용할수 있다.

공개원천-현대조작체계의 밀접한 련관

Linux가 성공하게 된것은 일반공개허가증(General Public License : GPL)때문이다. 하지만 공개원천이나 열린쏘프트웨어의 개념은 흔히 말하고 있는것처럼 실제적으로는 낡았다. 공개쏘프트웨어의 첫 제안자는 무료쏘프트웨어재단(Free Software Foundation : FSF)의 리챠드 스톨만이였다.

그는 자기가 작성한 몇가지 우수하고 현재 널리 보급된 쏘프트웨어에서 Emacs편집환경과 c언어나 c++언어용gcc콤파일러를 위한 GPL을 제안하였다. GNU수단의 충분한 목록에 대해서는 www. gnu. org를 보라. 리챠드 스톨만은 또한 GNU Hard라고 부르는 대상과제인 완전GNU조작체계에 대한 연구를 오래동안 진행하여 왔다. 10년가까이 개발해 왔으나 이 OS는 여전히 현실성이 없었다. 그후 수많은 유능한 기술자들이 이 대상과제에 힘을 집중하였고 그 과정에 이 대상과제는 Linux, BSD와 기타 다른 OS에 대한 개발방법을 찾아 내게 되였다. 이 조작체계들은 믿음성과 성능이 높은 공개원천으로 된다는데 리로운 점이 많다.

Linux의 믿음성문제는 크게 보면 코드를 해석하고 그것을 개선하며 또 그것을 변화

시키면서 실행시켜 보는 수백, 수천의 개발자들의 정교한 조사로부터 시작되였다. Eric Raymond가 자기의 론문 "Cathedral and the Bazar"에서 서술한바와 같이 《충분히 큰 베타테스터와 공동개발자기지가 주어 지면 거의 모든 문제는 빨리 해석되며 누구에게나 명백한것》으로 될것이다.

따라서 Linux기초코드를 조사하는 과정에 수많은 의문들이 제기된것으로 하여 어떤 닫겨 진 모형의 쏘프트웨어개발조직이 제공하는것보다 더 좋은 질담보(QA)를 얻을수 있게 한다. 이것은 당연히 더 질 좋은 프로그람을 만들어 내게 한다.

열린공개원천과 같은 단순한 개발모형은 그자체로 적당한 설계와 코드화방법으로 바꿀수 없지만 길이가 길어 지기때문에 배정된 길이를 초과한다.

Linux의 력사

성공한 수많은 력사와 마찬가지로 Linux도 필요성으로부터 제기된 대상과제의 하나로서 시작되였다.

1991년에 핀란드의 헬싱키종합대학의 대학생이였던 리누스 토발즈는 소편상에서 가상기억관리를 지원하는 첫 인텔 CPU인 i386에 기초한 PC를 자체로 준비하였다 (1991년).

MS-DOS조작체계에 완전하게 만족을 느끼지 못한 그는 MS-DOS대신 자기의 PC에 Minix조작체계를 실현할것을 결심하였다.

그는 곧 자기가 연구하는데 필요한 함수들과 특성들을 준비하기 위하여 Minix에 힘을 넣었다. 그후 그는 Minix가 학술연구용OS로서는 너무 컸으므로 처음부터 조작체계를 만들것을 결심하였다.

토발즈는 또한 중요하게 Linus와 Unix를 합하여 Linux라는 이름으로 자기의 새로운 조작체계의 원천코드를 인터네트상에서 자유롭게 리용할수 있게 즉 공개원천으로 리용할수 있게 만들것을 결심하였다.

첫 판본 0.01은 1991년 8월에 인터네트로부터 자료를 받을수 있게 만들어 졌다. 같은 해 10월에 리누스는 판본 0.02의 유용성을 공식적으로 발표하였다. 이 판본은 이미 bash shell, GNU gcc콤파일러 그리고 다른 기본적인 편의프로그람과 같이 UNIX사용자구역(Unix-user-land)프로그람을 실행할수 있었으며 또한 그리 많지 않았지만 다른 기본적인 유용성도 실현할수 있었다.

이 대상과제의 공개원천성에 의하여 즉시에 리용할수 있는 원천코드를 가지고 모든 해커들과 PC애호가들이 코드를 보고 그것을 해석하기 시작하였다.

많은 사람들이 자기들의 의견을 리누스에게 보내기 시작하였으며 리누스는 《사무》참조Linux원천코드개발나무를 발족하였다.

그들이 보낸 코드에 대한 의견을 받으면서 리누스는 그 의견을 거의나 접수하지 않았으나 일부 사람들은 공동개발자로 되였다.

첫 제품의 판본이 공개되기전에 3년이상이나 이러한 방법으로 개발이 계속되였다.

1994년 3월에 판본 1.0이 리용할수 있도록 만들어 졌다. 그러나 이 판본은 여전히 이모저모로 산만하고 변덕스러워 잘 리용할수 없었다. Linux 1.0은 TCP/IP, SLIP 그리고 인

쇄기 등을 지원할수 있도록 특색 있게 구성되였으며 리용가능한 넓은 범위의 PC장비들을 지원할수 있는 구동기들을 충분히 갖추었다.

그후에 실질적으로 Linux바람이 불기 시작하였으며 세계의 여기저기에서 수백만의 애호가들이 체계를 리용하기 시작하였다. 좀 더 일찌기 1992−1993년경에 첫 Linux 《배포판》이 출현하였다.

완전히 기능적인 OS를 제공하는 제1차적방법으로서 배포판은 Linux핵심부, X윈도우즈체계 그리고 완전한 응용프로그람과 편의프로그람들을 포함하고 있었으며 그 종류는 수백가지에 달하였다.

배포판은 또한 《설치자》를 포함하며 설치자는 OS의 2진이메지를 준비하고 기동/차단서술과 모든 구성요소들의 호환성을 보장하고 서로 전환할수 있게 준비되였으며 문서로도 제공되였다. 오늘날에는 널리 배포되고 있는 RedHat, SuSE 그리고 Caldera와 같은 완전히 성공적인 수많은 배포판들이 나왔다.

현재 개발된것들을 소개한다.

1991.8	판본 0.01
1991.10	사무용공개 0.02
1993.11	핵심부 0.99를 포함하는 첫 슬래크웨어(Slackware)배포판
1994.3	판본 1.0
1995.6	처음으로 Alpha구성방식에 이식
1996.10	Debia Linux가 궤도에 있는 스페이스샤틀(space shuttle)에 리용
1999.1	판본 2.2.0
2001.1	Linux 2.4.0발표
2001.7	Linux 2.4.6발표

현재 Linux에 의하여 제공되는 기능

Linux는 1991년에 초라하고 보잘것 없는 상태에서 출발하여 지금까지 많이 갱신되여왔다. 공개원천에 대한 인식이 더 넓게 확대됨에 따라 보다 많은 사람들이 Linux핵심부와 부분체계에 기여하게 되였다.

동시에 수천개이상의 사용자구역프로그람들이 날을 따라 빈번히 보충되였다.

World Wide Web의 병행적인 증가에 따라 어떤 Web싸이트들은 Linux에서 리용할수 있는 새로운 쏘프트웨어판본들에 대하여 단독으로 매일 공개하는데 리용되고 있다. 현재 Linux는 Intel, Sparc, Alpha, Mips, Motorola 68000계렬, PowerPC를 포함하는 많은 가동환경에서 리용할수 있다.

초기에 Linux는 POSIX (Portable Operating System Interface휴대용조작체계대면부)표준과 일치되였다.

이러한 Linux의 유연성은 Linux에서 개발된 응용프로그람들과 다른 POSIX호환조작체계들에 아주 쉽게 이식될수 있게 하였다.

인텔(Intel)계렬 CPU에 기초한 체계들에서 Linux2진파일들은 iBSCS표준에 맞게 되여있다.

실례로 정적으로 련결된 프로그람이 Free BSD나 Solaris우에서 재콤파일이 없이 실행될수 있게 한다.

기술적으로 Linux는 다음과 같은 기능들을 제공한다.

▼ 다중사용자
- 다중프로쎄스(Multi−process), 다중처리기(SMP)
- 프로쎄스조종
- POSIX형식의 말단관리
- TCP/IP, Ipv4, Ipv6
- 광범하고 다양한 하드웨어지원
- 선택적LRU알고리듬과 페지coloring에 의한 페지화요구
- 페지교환법(swapping)
- 캐쉬
- 동적 및 공유서고

▲ 각이한 파일체계(ext2fs, UFS, NTFS, HPFS, MS−DOS ISO9660, Coda 등)의 지원

핵심부 2.4의 새로운 기능

Linux 2.2는 Linux 2.0과 Linux 1.x계렬에 비하여 상당한 정도로 개선되였다. 이 체계는 새로운 파일체계들을 지원하고 새로운 파일에 대한 캐쉬체계를 가지고 있었으며 확대 및 축소가능하였다. Linux 2.4는 이러한 개선된 측면들로 구성되여 있으며 지금도 여전히 여러가지 다양한 환경에서 더 좋은 Linux의 핵심부로 발전하고 있다.

가상파일체계(VFS)층도 역시 초기의 Linux판본들로부터 커다란 변화를 이룩한 측면의 하나이다.

Linux 2.2의 수많은 놀라운 변화가 이 VFS계층에 의하여 특징 지어 지는데 이 VFS는 캐쉬의 성능을 더욱 높여 총체적으로 체계의 성능을 보다 효과적으로 향상시키고 있다.

그러나 Linux 2.2에서 체계는 아직도 Linux 2.4시대에 와서야 해결된 적지 않은 중요한 제한성을 가지고 있었다.

Linux 2.2가 체계를 조종하는 방법에서 한가지 중요한 제한성은 캐쉬에 대하여 2개의 완충기를 사용하는것이였다.

즉 입력용완충기와 출력용완충기를 사용하였다. 쉽게 생각해 볼수 있는것처럼 이것은 핵심부개발자들이 필요할 때마다 이 캐쉬들이 동기상태에 있는가를 항상 확인한후 코드화를 해야 하므로 작업이 매우 복잡해 지군 하였다.

Linux 2.4에서는 다중캐쉬체계를 없애고 모든 작업을 단일한 캐쉬층에서 진행하게 함으로써 이러한 복잡성을 완전히 없애 버리였다.

이러한 변화는 Linux 2.4를 보다 효과적으로 되게 하였으며 이로 하여 코드는 개발자들에게 더 리해하기 쉬워 지고 캐쉬에 요구된 기억의 크기도 크게 두개로 나누어 지게 되였다.

이러한 재기록과정에 많은 경쟁조건(보호되지 않은 변수에 대한 접근시 다중처리《경쟁》에서 발생되는 오유)이 해소되였으며 코드는 최신식의 체계로 확대축소가능하게 할수 있게 간결해 졌고 다중기록권(Multiple Volumes)을 포함하는 디스크의 속도는 휠씬 더 빨라 지게 되였다.

Linux의 핵심부는 장치구동기, 규약 그리고 다른 형태의 구성요소들을 포함하는 모듈화된 요소와 부분체계들의 집합이다. 이 모듈화된 부분체계들은 Linux핵심부를 확장할수 있는 표준적인 방법을 제공하는 응용프로그람작성대면부(API)에 의하여 Linux의 핵심부에 추가된다. 이 핵심부의 대부분의 내용들은 Linux에서 가장 중요한 동작을 하는 Linux조작체계구성요소들에 주목되고 있다. 핵심부 2.4계렬에서 Linux는 수많은 프로쎄스와 과제를 조종할수 있도록 그 능력이 방대하게 개선되였다. 과제의 한계는 현재 4090개 이상으로 높일수 있게 되였다. 게다가 과제관리기(scheduler)의 효과성이 상당히 개선되였는바 Linux 2.4는 그 이전의 판본들보다 더 많은 병행프로쎄스들을 더 훌륭히 조종할수 있게 되였다.

Linux 2.4는 또한 이전의 Linux핵심부에서 조종할수 있었던것보다 휠씬 더 많은《엔터프라이즈급》의 하드웨어들을 조종할수 있다.

실례로 Linux 2.4는 정교한 검사수정(patch)들로 4GB이상의 RAM을 주소화할수 있다.

어떤 기계에서는 12GB의 RAM을 배치구성할수 있으며 한개의 상주주소화공간의 크기가 8.3GB되는것도 있다.

SMP체계우에서의 축소확대기능은 현재 독자적지위에 있는 조작체계들과 비슷하며 어떤 경우에는 그것들을 릉가하고 있다.

다른 가동환경(platform)을 지원하는 기능도 역시 높아 지고 있다. 전통적인 구성방식에 이어 Linux 2.4는 현재 Transmeta Crusoe Processor를 직접 지원한다. 또한 3Com Palm Pilot뿐아니라 Psion5 등은 모두 특정방식에서 Linux를 실행할수 있다. 새로운 인텔IA64(다음세대 인텔i86계렬의 64bit처리기)는 아직 직접적으로 핵심부에 포함되지 않고 있다.

새로운 파일체계지원기능이 계속 보충되는 한편(Irix efs파일체계와 DVD디스크에 리용되는 UDF표준) 다른것들은 계속 쓸모가 없어 진다(QNX나 ext1).

저준위중단을 조종하는 새로운 혁신적방법의 하나인 tasklet의 도입은 보다 많은 TCP/IP탄창을 효과적으로 제공한다.

표준 DECNet를 조종할수 있는 새로운 망규약프로그람이 추가되였다.

Linux 2.2와 Linux 2.4에는 쟈바(Java)응용프로그람이 실행될 때마다 쟈바인터프레터(존재하는 경우)가 출발하는 내장지원기능이 포함되여 있다(Linux는 핵심부준위에서 이것을 실현할수 있는 첫 조작체계의 하나였다.).

Linux 2.4는 여전히 필요할 때마다 쟈바인터프레터의 적재를 지원하는 기능을 포함하고 있지만 정의된 쟈바구동기는 제거되기때문에 사용자는《Misc》구동기를 리용하기 위하여 그것의 배치구성(configuration)을 갱신해야 한다. Linux 2.4핵심부는 핵심부Web데몬(kernel web daemon)이나 혹은 kHTTPd를 포함한다.

이러한 기술은 정적인 Web페지를 보다 효과적으로 조종할수 있게 한다. 이 장의 서두에서 우리는 우에서 서술한것처럼 Linux에서 개선되고 강화된 모든 부분들과 상세한 측면들을 고찰하게 될것이며 그것에 포함되여 있는 핵심부의 동적특성을 알게 될것이다.

제2장. 핵심부의 콤파일

이 책에서 론의되는 많은 파일체계들은 핵심부(Kernel)를 재콤파일할것을 요구한다.

어떤 경우에는 우선 핵심부를 새로운 파일체계쏘프트웨어로 대치할것을 요구한다 (JFS, ReiserFS, XFS).

그러므로 여기서는 요구되는 기능들로 새로운 핵심부를 어떻게 만드는가를 총체적으로 리해하는것이 아주 중요하다. 새로운 핵심부를 콤파일하고 설치하는 작업은 많은 사람들에게 조작을 틀리게 하여 콤퓨터를 기동하지 못하게 될것 같은데로부터 위험하고 조심스러운 일처럼 보이고 있다. 모든 체계관리기과제에서와 같이 부정확한 방법으로 어떤 작업을 진행하면 무슨 사고가 날지 모른다. 그러므로 설치작업을 한 다음 새로운 핵심부를 콤파일하는것이 제일 좋다.

만일 어떤 작업을 틀리게 하면 체계는 시간을 많이 소모함이 없이도 스크래치로부터 재설치될수 있다.

틀린 작업을 계속하면 나중에 보충적쏘프트웨어를 설치한후(자료기지관리기와 같은) 더 큰 고통을 받게 될것이다. 그러나 걱정할 필요는 조금도 없다. 핵심부의 콤파일과 설치는 생각한것보다는 훨씬 단순하다. 표본핵심부를 콤파일하는 단계를 다음에 보여 준다.

원천코드의 나무구조

이 책을 보다 더 잘 리해할수 있을뿐아니라 핵심부를 검사수정하고 그것을 콤파일할수 있도록 하기 위하여서는 원천코드의 나무구조에 대한 표상을 잘 가지는것이 중요하다.

```
|    |
|    \--- video4linux
|          \--- bttv
|--- arch
|       \--- i386
|              |--- boot
|              |       |--- compressed
|              |       \--- tools
|              |--- kernel
|              |--- lib
|              |--- math-emu
|              \--- mm
|--- configs
|--- drivers
|      |--- acorn
|      |      |--- block
|      |      |--- char
|      |      |--- net
|      |      \--- scsi
|      |--- ap1000
|      |--- block
|      |       \--- paride
|      |--- cdrom
|      |--- char
|      |      |--- ftape
|      |      |       |--- compressor
|      |      |       |--- lowlevel
|      |      |       \--- zftape
|      |      |--- hfmodem
|      |      |--- ip2
|      |      \--- joystick
|      |--- dio
|      |--- fc4
|      |--- isdn
|      |      |--- act2000
|      |      |--- avmbl
|      |      |--- divert
|      |      |--- eicon
|      |      |--- hisax
```

```
|    |    |--- icn
|    |    |--- isdnloop
|    |    |--- pcbit
|    |    \--- sc
|    |--- macintosh
|    |--- misc
|    |--- net
|    |    |--- fc
|    |    |--- hamradio
|    |    |    |--- soundmodem
|    |    |    \--- irda
|    |--- nubus
|    |--- pci
|    |--- pnp
|    |--- sbus
|    |    |--- audio
|    |    \--- char
|    |--- scsi
|    |    \--- aic7xxx
|    |--- sgi
|    |    \--- char
|    |--- sound
|    |    \--- lowlevel
|    |--- tc
|    |--- usb
|    |    \--- maps
|    |--- video
|    \--- zorro
|--- fs
|    |--- adfs
|    |--- affs
|    |--- autofs
|    |--- coda
|    |--- devpts
|    |--- efs
|    |--- ext2
|    |--- fat
|    |--- hfs
```

```
|    |--- hpfs
|    |--- isofs
|    |--- lockd
|    |--- minix
|    |--- msdos
|    |--- ncpfs
|    |--- nfs
|    |--- nfsd
|    |--- nls
|    |--- ntfs
|    |--- proc
|    |--- qnx4
|    |--- romfs
|    |--- smbfs
|    |--- sysv
|    |--- ufs
|    |--- umsdos
|    \--- vfat
|--- ibcs
|    |--- Doc
|    |--- PROD.  Patches
|    |--- Patches
|    |--- Tools
|    |--- VSYS
|    |--- devtrace
|    |--- iBCSemul
|    |    \--- maps
|    |--- include
|    |    \--- ibcs
|    \--- x286emul
|--- include
|    |--- asm -> asm-i386
|    |--- asm-generic
|    |--- asm-i386
|    |--- linux
|    |       |--- byteorder
|    |       |--- lockd
|    |       |--- modules
|    |       |--- modules-BOOT
```

```
|        |--- modules-smp
|        |--- modules-up
|        |--- nfsd
|        |--- raid
|        \--- sunrpc
|--- net
|        \--- irda
|--- scsi
\--- video
--- init
--- ipc
--- kernel
--- lib
--- mm
--- modules
--- net
    --- 802
        |--- pseudo
        \--- transit
    --- appletalk
    --- ax25
    --- bridgs
    --- core
    --- decnet
    --- econet
    --- ethernet
    --- ipv4
    --- ipv6
    --- ipx
    --- irda
        |--- compressors
        |--- ircomm
        |--- irlan
        \--- irlpt
    --- lapb
    --- netlink
    --- netrom
    --- packet
    --- rose
```

Linux가 구성방식의존코드와 비의존코드로 어떻게 구성되여 있는가를 잘 아는것이 중요하다. 핵심부원천의 95%는 비의존코드이며 따라서 Linux의 모든 이식묶음과 정확히 같다. 나머지 5%는 보통 아쎔블러코드나 시계박자주파수와 같은 작고 상세한 것들이다.

arch/등록부

이 등록부는 구성방식에 의존하는 코드가 위치하는 곳이다.

이 등록부아래에는 Linux의 매 이식묶음에 대하여 3개이상의 부분등록부(kernel/, lib/, mm/)들이 존재한다.

kernel부분등록부는 신호조종, 시계조종 등과 같은 일반 핵심부특유의 구성방식의존 실현부가 포함되여 있다.

lib부분등록부에는 구성방식의존원천코드로부터 콤파일되면 더 빨리 실행되는 국부적인 서고함수들이 차례로 포함되여 있다.

mm/등록부는 국부기억조종실현부를 포함한다.

drivers/등록부

모든 구동기의 원천코드는 이 등록부에 있다. Linux 2.4가 지원하는 장치의 종류가 아주 다양하므로 이 등록부에는 많은 원천코드들이 포함되는데 실례로 모든 핵심부원천코드의 50%이상을 차지한다.

fs/등록부

fs/등록부는 파일체계를 지원하는 코드들을 모두 포함하는 등록부이다. IBM의 JFS나 Hans Reiser의 ReiserFS와 같은 새로운 파일체계를 실현하려는 사람들은 해당 파일체계를 위한 원천코드들을 포함할수 있도록 이 등록부를 검사수정하여야 한다.

include/등록부

새로운 핵심부를 실제적으로 콤파일하기전에 반드시 배치구성이 되여야 한다. 배치구성(configuring)은 어느 구동기나 속성 그리고 어느 모듈을 핵심부에 콤파일할것인가를 제작편의프로그람에 통보해 준다. 대다수의 표준배포판들은 암시적으로 단일처리기용핵심부로 되여 있다. 핵심부를 표준 SMP속성으로 실현하려면 SMP에 따르는 배치구성이 요구된다.

ipc/등록부

프로쎄스들간 통신을 조종하기 위하여 필요한 모든 코드들이 이 등록부에 보관된다. 중요한 모든 쎄마포-조종용C코드(sem. c)도 바로 여기에 포함된다.

그럼에도 불구하고 이 등록부는 코드의 크기가 3751행밖에 되지 않는다.

init/등록부

fork()를 실현하기 위한 코드와 흔히 실행되는 코드부분으로서 cpu_idle() loop와 같은 많은 중요한 코드를 포함하는 Main.c가 바로 init/등록부에 있다.

기동할 때 bogomips를 생성하는 코드가 여기에 존재하는데 그 코드들은 처리기의 속도지시를 관측한다. 실제적인 처리기의 속도는 측정하지 못한다.

```
void_init calibrate_delay(void)
{
     unsigned long ticks, loopbit;
     int lps_precision = LPS_PREC;

     loops_per_sec = (1<<12);

     printk(“Calibrating delay loop... ”);
```

```
    while (loops_per_sec << = 1) {
         /* wait for "start of" clock tick */
         ticks = jiffies;
         while (ticks = = jiffies)
         /* nothing */;
         /* Go . . */
         ticks = jiffies;
         _delay(loops_per_sec);
         ticks = jiffies – ticks;
         if (ticks)
            break;
}
```

lib/등록부

이 등록부에는 종종 핵심부의 다른 부분에서 요구되는 코드들이 존재한다. 실례로 inflate.c를 여기서 찾아 볼수 있다.

이 코드는 기동시에 핵심부의 압축을 해제하고 그것을 기억에 적재시킨다.

표준PKZIP압축알고리듬을 리용하여 어떻게 압축을 해제하는가에 대해서는 이미 알고 있을것이다.

kernel/등록부

이 등록부에는 제일 빈번히 호출되는 핵심부함수들중에서 그 일부가 존재하고 있다. 스케쥴러(Scheduler)외에 fork()와 timer.c도 여기에서 찾아 볼수 있고 printk.c도 이 등록부에서 찾을수 있을것이다.

핵심부코드전반에 걸쳐 printk()는 printf()함수가 핵심부로부터 호출될 때에는 SMP기능을 수행하지 못하기때문에 printf()대신에 printk()를 리용한다.

따라서 서로 다른 CPU우에서 동작하는 여러가지 과제들은 동시에 출력을 보장하면서 조종대나 체계가동일지등록부에 써넣을수 있다.

mm/등록부

mm은 기억관리(memory management)를 의미한다.

이 등록부는 Linux핵심부에서 가상기억관리를 실현하기 위한 원천코드들을 포함한다.

net/등록부

TCP/IP, Netware, Appletalk와 같이 망을 지원하는 모든 코드들은 이 등록부에 보관한다.

핵심부의 콤파일

실천적으로 핵심부를 콤파일하기에 앞서 콤파일편의프로그람에 어느 기능이 요구되며 구축된 그러한 기능들을 핵심부에 포함시키겠는지 혹은 그것들을 적재가능한 모듈로 배치구성하겠는가를 알려 주어야 한다(핵심부가 적재가능한 모듈들을 어떻게 관리하는가에 대해서는 다음에 고찰한다.).

다음 표는 핵심부를 콤파일하는데 어떤 명령들이 사용되는가를 보여 준다.

형	명령(root)
Text prompt Text menus(ncurses style) GUI (X실행요구)	make config make menuconfig Make xconfig

Make config는 선행한 선택항목을 기억하므로 언제나 그것을 다시 출발시킬수 있으며 필요할 때마다 변경시킬수 있다. 핵심부가 만족스럽게 배치구성되면 콤파일처리를 진행할수 있다.

```
root@maguro/usr/src#make dep; make clean;
make bzImage; make modules-install
```

명령 "make bzImage"는 핵심부를 콤파일하고 arch/i386/boot등록부에서 bzImage를 호출한 파일을 남겨 둔다. 우의 명령을 단계적으로 진행하는데는 일정한 시간이 걸린다.

512MB RAM을 가진 2중 PⅢ 700MHz CPU상에서 이 명령은 약 4min정도 걸리는데 속도가 더 뜬 콤퓨터에서는 더 오랜 시간이 걸린다. 이제 새로운 핵심부를 설치해 보겠다. 많은 사람들이 핵심부설치에 LILO(Linux Loader : Linux적재기)를 리용한다.

명령 make bzlilo는 핵심부를 설치하고 그우에서 LILO를 실행시키며 기동에 필요한 모든 준비를 갖출수 있게 한다. 그러나 이 조작은 LILO가 해당 체계에 다음의 방법으로 배치구성될 때에만 가능하다. 즉 핵심부 /vmlinuz가 /sbin에 있으며 lilo config (/etc/lilo.conf)가 이것과 호환되는 경우에만 가능하다.

다른 한편 lilo등록부도 리용해야 한다. lilo를 리용하여 설치작업을 진행하는것은 아주 명백하고 쉬운 프로그람이지만 그 배치구성파일을 가진 사람들은 혼돈될 경향성도 있다. 이제 config(/etc/lilo/config(변경된 판본)이나 혹은 /etc/lilo.con(새 판본))파일을 보고 현재의 설치가 어떤 동작을 수행하는가를 알아 보자.

Config파일은 아래와 같이 되여 있다.

```
image=/vmlinuz
    lable=Linux
    root=/dev/hda1
    …
```

“image=”는 현재 설치된 핵심부에 설정된다. 많은 사람들은 /vmlinuz를 리용한다. “lable”은 기동할 핵심부가 조작체계를 결정할수 있도록 LILO가 리용하는 기호이며 “root”는 특정한 조작체계의 /등록부이다(뿌리등록부).

변경된 핵심부의 여벌복사(bakcup)를 진행하고 지정한 장소에 생성해야 할 bzImage를 복사한다(만일 /vmlinuz를 리용하면 cp bzImage/vmlinuz로 할수도 있다.).

LILO를 재실행하고 핵심부를 기동관리기(boot manager)에 첨가한다. 다른 한편 새로운 핵심부를 LILO배치구성에 실제적으로 넘겨 주기전에 검열을 위한 기동디스크를 만들려고 하면 많은 배포판들과 함께 배포되는 mkbootdisk편의프로그람을 리용할수 있다.

GNU gcc콤파일러

Linux핵심부는 GNU gcc콤파일러용으로 서술된다. 그러므로 어떤 다른 C콤파일러를 가지고 핵심부를 콤파일하려고 하면 완전한 오유를 산생시킬수 있다.

원천코드는 아무 콤파일러나 리용하는 현상을 막고 gcc콤파일러를 사용하려는 의미에서 완전한 gcc정의명령들로 구성된다. 이러한 gcc와의 관계때문에 핵심부원천코드는 다른 C콤파일러를 개발리용하는 사용자들에게는 좀 이상한 감촉을 가지게 한다. 이러한 느낌을 없애는 하나의 공통적인 특수한 표현법이 “inline functions”(직결기능)를 리용하는 것이다.

직결기능은 단일한 함수 즉 호출함수의 참조시 매번 함수호출을 실행할 대신 그 매개를 재현하는 방법으로 호출함수를 완전히 확장하는 gcc콤파일러에 대한 명령이다. 어떤 경우에는 이것이 이식가능한 코드를 얻는것을 불가능하게 하는 객체도 될수 있다. 그러나 이것은 사실상 경우에 맞지 않는다. Linux의 gcc콤파일러는 Linux가 이식될수 있는 모든 가동환경에 존재하기때문에 코드는 이러한 구성방식으로 얼마든지 이식할수 있다. 물론 다른 C콤파일러에서 정확하게 콤파일될수 있다는 견지에서는 이식이 불가능하다. 다시 말하여 핵심부가 구성방식과 콤파일러를 초월하여 이식성을 요구한다는것은 아니다. Linux핵심부는 믿음성과 효과성을 목적으로 최량화되며 이 두가지 목적은 두말할 여지없이 아주 중요한 내용으로 된다.

코드화관례

이 책을 읽은후에는 Linux핵심부에 기여할 생각이 있으리라고 본다. 그 어떤 관례를 존중하는 조건에서는 설명이 항상 /* */의 기호로 표현되며 이것은 한행에 대해서도 허용된다. //의 해설표시는 허용되지 않는다.

흔히 함수에 리용되는 열린괄호 ({)는 분리행우에 있게 된다.

명령문은 다음과 같은 방법으로 코드화된다.

```
If(str[0]>='0' && str[0]<='9'){
       Strcpy(name,“ttys”);
```

```
Strncpy(name+4, str, sizeof(name)-5);
} else
  strncpy(name, str, sizeof(name) - 1);
name[sizeof(name)-1]=0;
```

명령문인 경우에는 단일행도 허용된다.

if (! strcmp(str, “ttya”)) strcpy(name, “ttys0”) ; ne ifs are quite ok ;

이전시기부터 이미 핵심부원천코드에는 항상 goto문이 많이 리용되였다. Linux도 례외가 아니다. Linux도 약 80개 행의 코드마다에 goto문이 한개정도 포함된다. 이것은 프로그람작성에서 속도가 떠지게 하는것이 아니라 오히려 속도가 빠른 코드를 작성하기 위하여 제기된 요구이기도 하다. 반복화된 순환명령문에서는 정지명령문보다 오히려 코드를 줄이기 위하여 goto를 리용하는것이 훨씬 쉽다.

구성방식의존성

Linux는 광범하고 다양한 처리기가동환경에서 실행된다. 거의 매달 다른 구성방식에 대하여 새로운 이식을 성공적으로 진행한 자료들이 보고되고 있다. Linux가 i386처리기우에서 시작되였다는데 대하여 강조한다.

이것은 핵심부코드의 어느 곳에서나 명백히 찾아 볼수 있다.

다음의 표는 지금까지 달성한 이식상태를 보여 준다.

처리기형	비트길이	주장자
Intelx86	32	리누스 토발즈
Crusoe	32	리누스 토발즈
MIPS	32/64	알란 콕스
IA64	64	트릴리안 대상과제
PA-RISC	64	푸핀그룹
Alpha	64	리챠드 헨더슨
ARM	32	루쎌킹
Sparc	32/64	데이비드. 에쓰. 밀러
PPc	32	코트 다우겐
M68000	32	제스 쏘렌센

제3장. 파일체계란 무엇인가

이 장에서는 파일체계의 구성성분들이 무엇으로 이루어 지며 그것들이 어떻게 Linux 2.4에서 실현되는가에 대하여 고찰한다. 일반적으로 다른 UNIX파일체계에서 설정한 대부분의 내용들이 Linux에서도 역시 유효하다. 그러나 일부 경우에는 Linux가 오히려 UNIX보다 구성방식비의존성, 속도, 효과성, 안전성 그리고 실현의 정교성을 보다 쉽게 할수 있는 여유를 가지고 있다.

파일체계의 일반적형식

이 장의 구성부분은 오스트랄리아의 네일즈 브라운의 Linux가상파일체계에 대한 중요한 문서를 기초로 하여 이루어 졌다.

파일체계는 콤퓨터의 하드디스크가 국부디스크든지 망을 리용하는 기록권이든지 혹은 저장망(SAN)의 외부공유디스크든지 관계없이 자료를 기억시키거나 검색하기 위한 조작체계의 론리적수단이다. 특히 파일체계는 UNIX양상의 OS에 요구되는 기본적인 조작을 실현하고 있다.

▼ 파일을 생성하거나 삭제한다(즉 기억매체우에서 공간의 배정 및 해제).
- 읽기 및 쓰기를 위한 파일의 열기
- 파일안에서의 찾기(주의 : Linux는 파일레코드의 표기를 위한 핵심부준위에서의 지원은 제공하지 않는다.)
- 파일의 닫기
- 파일의 그룹을 구성하기 위한 등록부의 생성
- 등록부내용의 목록표시

▲ 등록부로부터 파일의 제거

이러한 기능들을 현대적인 UNIX환경으로 우리가 일반적으로 알게 되기까지에는 여러해의 발전과정을 거치였으며 복잡한 파일관리능력과 자료관리대면부의 값 눅은 모임으로 제공되고 있다.

처음부터 Linux는 사용자들이 하나이상의 파일체계를 쓰는데 적응시키려고 노력하여 왔다. 사실 사용자가 DOS나 FAT32와 같은 다른 디스크우에 하나이상의 Linux구획이 아닌 구분구역을 가지게 하는것은 일반성이 없었다. Linux우에 이 기록권(volume)들을 올려태우기(mount)하려고 하면 그 핵심부에 적재된 DOS나 FAT32파일체계가 있어야 하였다. 그러므로 Linux개발자들이 핵심부와 잠재적으로 많은 파일체계실현사이의 호상관계를 단순화하려고 한것은 론리적이였다. 따라서 그들은 Linux가상파일체계(VFS)를 생성하게 되였다. Linux VFS는 ext2, DOS, FAT32, OS/2, HPFS와 다른 체계에서와 같은 실제적인 파일체계우에 있는 추상적인 층이다. Linux VFS에 의하여 사용자프로그람들은 구획우

에서 리용되는 파일체계에 대한 구체적특성들을 알지 못해도 되였다.

즉 체계호출이 항상 같았기때문에 기본기능(지우기, 복사, 생성)은 언제나 동일하였다.

다음에는 VFS기본기능들을 적응시키고 그것들을 파일체계의 규약과 정의에 따라 수행하게 하는것이 파일체계실현의 중요과제였다.

그러므로 Linux VMS는 DOS, OS/2, FPFS나 JFS*[1]구획에 관계없이 파일을 편집하기 위한 표준 "Vi mgfile.foo"을 생성해 낼수 있게 하였다. 이제 파일체계에 대한 폭 넓은 개념을 다시 고찰해 보겠다. 파일은 단순히 요소들의 순서화된 렬이며 여기서 요소들은 장치적실현에 따라 기계단어들로 될수 있고 문자나 혹은 비트들로 될수도 있다. 프로그람이나 사용자는 파일체계를 리용해서만 파일을 생성하거나 변경하거나 지울수 있다.

일반적으로 UNIX계렬에 대해서는 파일체계의 준위에서 파일이 양식화*[2](format)되지 않고 있다. 모든 양식화는 보다 높은 준위의 모듈 혹은 요구한다면 사용자지원프로그람에 의하여 실현된다. 특정한 사용자에게 있어서 한개 파일은 하나의 이름을 가지며 그 이름은 기호적표기로 된다(기호적이름은 어떤 한계까지의 임의의 길이로 될수 있으며 그 자체의 고유한 문장구조를 가진다.).

사용자는 기호적파일이름과 파일안의 요소에 대하여 선형적인 첨수를 정의하여 파일안의 요소를 참조할수 있다.

고준위모듈을 리용하여 사용자는 역시 문맥에 따라 직접적으로 적당히 정의된 요소의 렬을 참조할수도 있다.

등록부(directory)는 파일체계에 의하여 유지보존되며 입구점(entry)의 목록을 포함하는 특수한 파일이다.

사용자에게는 입구점이 파일로 나타나며 그것은 기호입구점이름에 의하여 접근되는데 이때 기호입구점이름은 사용자의 파일이름으로 된다.

입구점이름은 그것이 생성되는 등록부안에서만은 유일해야 한다. 그러므로 UNIX에서와 Linux에서 파일이름공간의 범위는 한개 등록부로만 간주된다. 실제적으로 매개 입구점은 지적자로 구성되는데 지적자의 종류는 2가지이다.

입구점은 어느 한 파일을 직접적으로 지적하거나(파일 그자체가 등록부일수도 있다.) 아니면 같거나 혹은 다른 등록부안의 어떤 다른 입구점을 지적할수도 있다.

다른 등록부입구점을 지적하는 입구점을 련결(link)이라고 부른다. 련결과 관련된 정보만은 그것이 련결하는 입구점에 대한 지적자이다. 이 지적자는 계층안에 련결된 입구점을 유일적으로 식별하는 기호이름에 의하여 정의된다.

련결은 그것이 효과적으로 지적하는 가지로부터의 접근조종정보를 관리한다.

---

*[1] JFS는 IBM실행기록파일체계이다.

*[2] 다른 많은 조작체계들은 파일내용의 양식화를 위한 자체의 표식을 가지고 있다. 실례로 IBM OS/390에는 고정된 레코드와 변하는 레코드에 대한 서로 다른 양식이 있다. UNIX는 단순성을 위하여 파일내용의 양식을 파일체계가 알지 못하게 하는 방법을 택하고 있다.

파일의 계층구조

한 등록부안에서 직접적으로 지적된 파일은 그 등록부에 대하여 직접하위이다(이때 지적한 등록부는 파일에 대한 직접상위이다.).

그자체가 두번째 등록부에 대하여 직접하위인 등록부에 대하여 또 직접하위인 파일은 두번째 등록부에 대하여서도 하위이다(류사하게 두번째 등록부는 파일에 대하여 상위이다.).

뿌리(root)는 령준위를 가지며 그것에 대하여 직접하위인 파일은 1준위로 된다. 개념을 확장하면 하위성(상위성)은 직접하위(상위)파일사슬을 거쳐 분할되는 준위들의 어떤 수에 의하여 정의된다(하위성에 따르는 준위수의 증가에 의하여 지장을 받는 사용자의 경우는 준위수를 부수로 할수도 있다.).

이때 련결은 독립적이기는 하지만 나무구조우에 덧붙여 지는것으로 고찰할수 있다. 상위성이나 하위성의 표기는 련결과는 아무런 관계도 없으며 오직 가지와만 관련이 있다는데 대하여 강조해 둔다. 이러한 종류의 나무계층구조에서는 사용자가 등록부를 빈번히 여기저기 움직이는것보다는 하나 혹은 몇개의 등록부에서 작업할수 있다. 따라서 류사한 관심을 가지는 사용자들은 공통적인 파일들을 공유할수 있고 또 자기에게만 요구되는 개인용파일을 가질수 있게 계층이 배렬되는것은 자연스럽다.

어느 한 순간에 한 사용자가 작업등록부라는 한 등록부안에서 조작하는 경우를 고찰한다. 사용자는 간단하게 입구점이름을 정의함으로써 작업등록부안의 입구점에 의하여 파일에 효과적으로 접근할수 있다.

하나이상의 사용자에 대해서는 어느 한 순간에 동일한 작업등록부를 가질수 있다. 련결이 없는 단순계층나무의 한 실례를 그림 3-1에 보여 주었다. 동그라미로 표시한
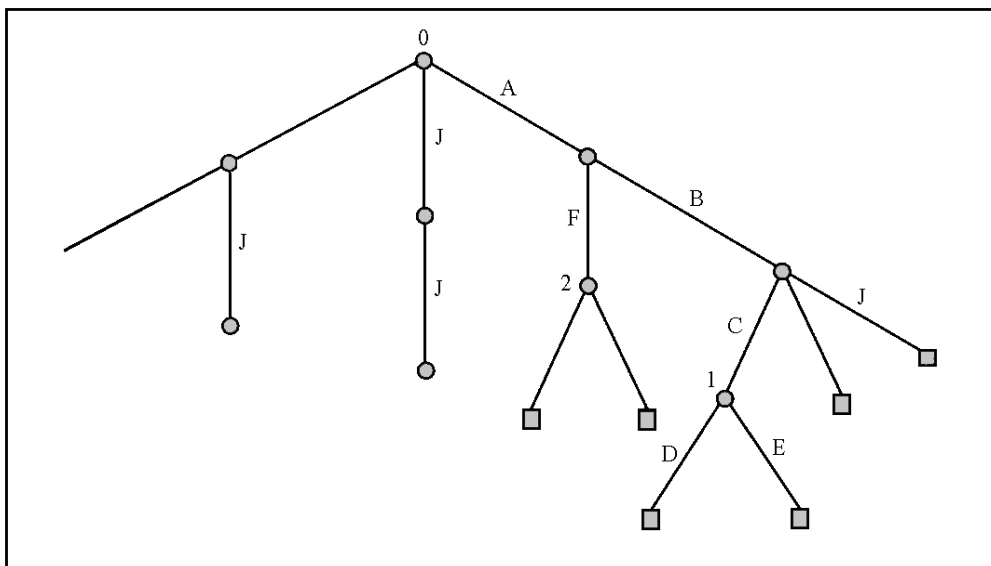
그림 3-1. 련결 없는 계층의 실례

비종단마디(끝이 아닌 마디)는 등록부파일을 지적하며 한편 그러한 매개 마디로부터 아래방향으로 향하는 선들은 그 마디에 대응하는 등록부에서의 입구점(즉 가지)을 지적한다. 4각형으로 표시한 종단마디(끝인 마디)는 등록부가 아닌 파일들을 지적한다.

문자들은 입구점이름을 가리키며 수자들은 그림에서 등록부들을 식별하기 위한 서술목적으로만 리용된다. 실례로 문자 “J” 는 그림안의 서로 다른 등록부들에서의 여러가지 입구점들의 입구점이름이며 수자 “0” 은 뿌리로 간주한다. 입구점이름은 그것이 생성되는 등록부에 관해서만 의미를 가지며 그 등록부밖에서는 유일할수도 있고 유일하지 않을수도 있다.

여러가지 리유로부터 전체로서 계층안의 입구점을 유일하게 정의하는 기호이름을 가지는것이 좋다.

그러한 이름은 뿌리와 관련하여 지어 주는데 보통 이것을 나무이름이라고 한다. 이것은 뿌리로부터 가지사슬을 거쳐 입구에 도달하기 위하여 요구되는 입구점이름사슬로 구성된다. 많은 경우 사용자는 입구점의 나무이름을 알 필요가 없다.

다른 방법으로 정립하여 표현하지 않는 한 파일의 나무이름은 뿌리와 관련시켜 정의한다. 또한 임의의 등록부에 관하여 유일하게 이름을 정의할수도 있다.

임의의 이름을 가진 련결은 명령에 의하여 다른 등록부에 설정될수도 있다.

ln-s 련결이름, 경로이름

경로이름에 련결할 입구점의 이름은 작업등록부와 관련된 나무이름으로 정의할수도 있으며 혹은 뿌리와 관련된 이름 혹은 보다 일반적으로 경로이름으로 정의할수도 있다(아래에 정의되였다.).

파일은 거기에 어떻게 접근하는가에 따라 서로 다른 사용자에 대하여 서로 다른 이름을 가질수 있다.

련결은 계층안의 어떤 곳에서나 가지에 대한 지름경로로 봉사하며 사용자에게 그 련결이 실제적으로 요구되는 파일을 직접적으로 지적하는 가지라는것을 알려 준다.

비록 련결들이 가지의 나무구조안에 이미 주어 진 가지에 대한 기본적인 능력을 전혀 추가하지 못한다고 해도 파일체계가 가지들을 리용하는 일을 아주 쉽게 해주게 된다.

련결은 또한 공유파일들의 2중복사조건도 상당히 감소시켜 준다.

그림 3-1의 나무구조에 련결들을 덧붙인것을 그림 3-2에서 설명하고 있다.

마디로부터 아래방향으로 향한 점선들은 다른 입구점과 련결한 입구점을 보여 준다. 련결들이 나무구조에 보충되면 방향그라프로 된다(물론 방향은 매 마디로부터 아래방향으로 향한다.).

그림 3-2의 실례에서 등록부 2에 G라는 이름을 가진 입구점은 등록부 3에 C라는 이름을 가진 가지와 련결된다. 등록부 4에 C라는 이름을 가진 입구점(등록부안을 제외하고는 이름이 꼭 유일할것을 요구하지 않는다는 사실을 상기)은 등록부 2의 입구점 G와 련결되며 따라서 등록부 3의 C에 대한 련결로서 작용한다.

이 련결들은 둘다 효과적으로 등록부 1을 지적한다. 련결들을 포함하는 나무이름과 류사한 이름을 가지는것이 좋다고 인정된다. 그러한 이름이 경로이름(pathname)이며 이것은
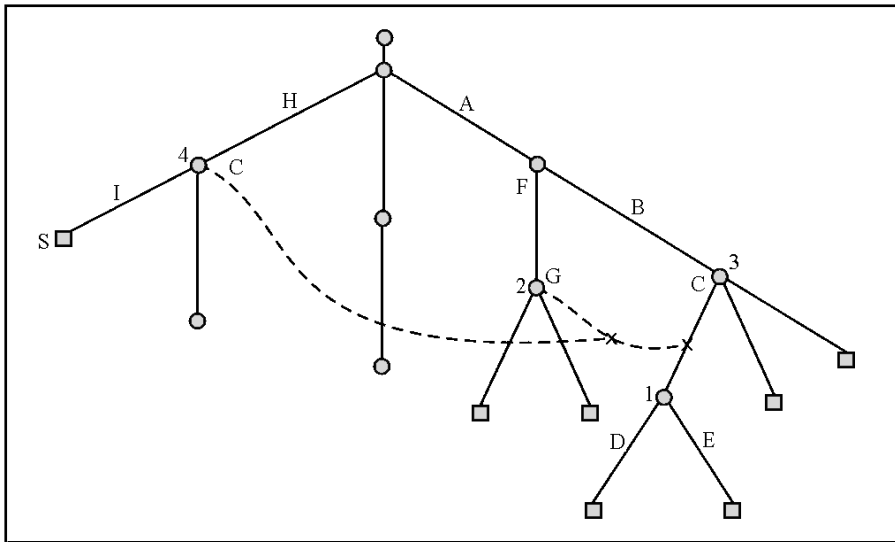
그림 3-2. 보충된 련결을 포함한 그림 3-1의 실례

다른 방법으로 특별히 정의하여 서술하지 않는 한 뿌리와 관계되는것으로 가정한다. 파일(뿌리와 관련되는)의 경로이름은 뿌리에 관하여 파일의 이름을 명명하는데 리용된 입구점이름의 사슬이다. 작업등록부는 항상 경로이름에 의하여 설정된다.

사용자는 아래와 같은 명령으로 작업등록부를 변경시킬수 있다.

cd 경로이름

여기서 경로이름은(변경된) 작업등록부 혹은 뿌리와 관련될수 있다. 뿌리와 다른 등록부에 관한 경로이름의 정의는 다음의 례외에 따르는 나무이름의 정의와 류사하다. 등록부에 대한 직접하위파일의 개념은 입구점에 의하여 효과적으로 지적되는 파일의 개념으로 바뀌운다. 또한 파일에 대한 직접상위등록부에 대한 개념은 우에서 본 효과적인 지적자 즉 등록부가 파일에 접근하기 위하여 이미 먼저 리용된 입구점에 의존하는 지적자의 거꿀형식으로 정의된 개념으로 바뀌운다.

일반적으로 임의의 파일은 현행작업등록부에 관한 경로이름(사실상 나무이름이나 입구점이름일수 있는)에 의하여 정의될수 있다. 파일은 또한 뿌리와 관련된 경로이름에 의하여서도 정의될수 있다. 앞의 경우에 경로이름은 두점으로 시작되며 뒤의 경우에는 그렇게 되지 않는다.

파일체계에서의 객체

우리의 경우에 파일체계는 객체를 담아 두는 저장통으로 고찰할수 있다.

여기에서 객체들은 파일체계가 사용자에게 추상화하여 제공한 모든 론리적실체로 된다. 파일체계에서 한가지 명백한 객체의 실례는 파일이며 다른것은 객체로 된다.

례하면 다른것들은 아직 기호적이름에 불과하며 사용자는 상위블로크, 색인마디와 같이 말단사용자의 눈에는 직접 보이지 않는 객체를 가지게 된다.

파일체계와 Linux와의 관계-가상파일체계

UNIX체계에서 많은 파일들은 수백, 수천 혹은 그이상으로 쉽게 배렬되기때문에 파일체계조종구조를 조작하는 과제가 응용프로그람과는 분리되며 엄밀하게 말하여 아직은 핵심부요소로 되지 않는다. Linux핵심부의 파일부분체계(file sub system)는 파일이 어떤것인가에 대한 특수한 추상적표상을 준다. Linux자체고유의 ext2 혹은 procfs, ReiserFS 그리고 다른 파일체계들에서 구체적인 내용은 알수 없고 모든 파일체계는 동일하게 취급된다. 핵심부는 매개 파일들을 서로 붙일수 있는 가장 기본적인 후크(hook)(우에서 언급된 VFS)만을 제공하며 중간조종구조의 생성과 관리를 통하여 실제적기능성(핵심부와 응용프로그람에)을 제공한다. 이 추상화준위는 모든 파일조종이 사용자공간파일체계조종코드보다는 오히려 실증된 조종구조객체에 작용하는 핵심부체계호출로서 실현될수 있게 하는 기본열쇠이다.

여기서 사용자공간파일체계조종코드는 파일을 열기 위하여 조종구조로 하여금 체계규모와 처리기 각각을 차례로 배정 혹은 해제할수 있게 한다. 응용프로그람은 파일을 어떤 입출력매체형태 대표적으로 SISC디스크와 같은 기억장치 등에 기억된 바이트들의 선형적인 배렬을 주소화하기 위한 추상화된 형태로 리용한다.

파일에 접근하기 위하여 OS는 매개 파일자료에 대하여 열기, 닫기, 읽기, 쓰기를 위한 파일관리대면부를 제공한다.

완충기, 캐쉬 및 기억기쪼각수집

완충기는 완충기억된 파일의 내용을 의미하는것이 아니라 물리적디스크에 완충기억된
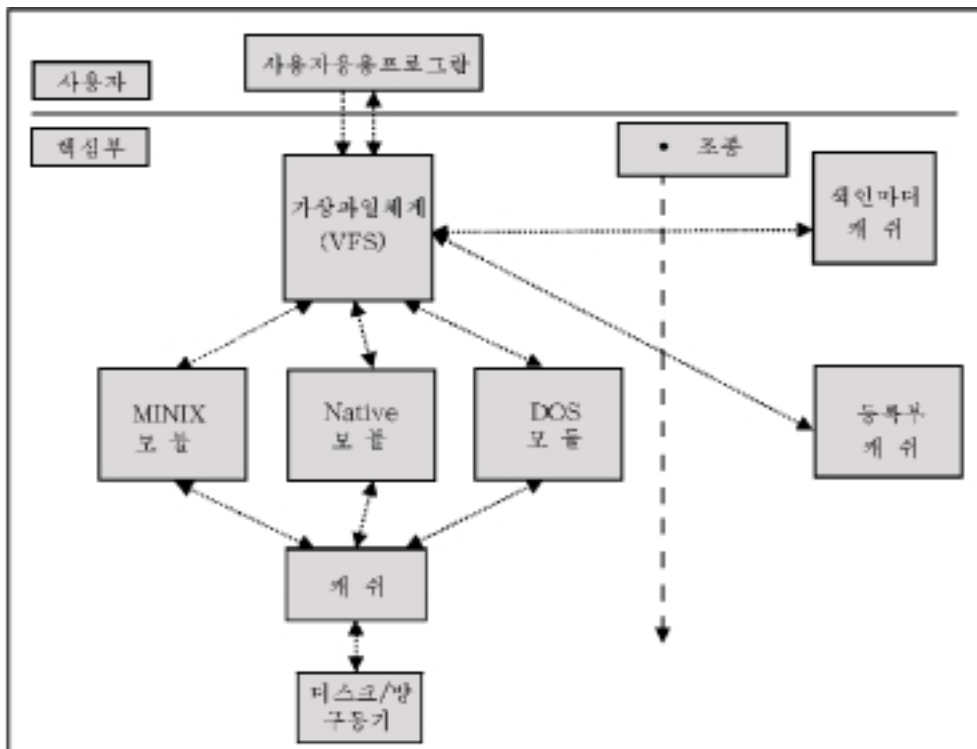
그림 3-3. VFS의 기본구조

내용을 표현한다. 반대로 캐쉬는 실제적으로 파일의 내용을 고속으로 기억한다.

Linux에서 완충기와 고속완충기의 동작은 가상기억기(VM)의 기능과 성능에 크게 의존한다. 가상기억기는 어떤 조작체계에서나 가장 복잡한 구성요소중의 하나이며 그 구체적내용에 대해서는 여기서 깊이 취급하지 않는다. Linux 2.4의 가상기억기에 대한 구체적내용은 2000년 6월 McGraw-Hill에서 출판된 《Linux internal》을 참조할수 있다.

그림 3-3은 VFS의 기본구조를 보여 준다.

그림은 핵심부가 주기억으로부터 어떻게 완충기를 배정하며 캐쉬를 어떻게 관리하는가를 보여 준다.

완충기의 캐쉬

파일체계가 사용자프로그람에 의하여 올려태우기되면 자료블로크를 읽고 쓰기 위하여 블로크장치에 대한 많은 요청을 생성한다. 모든 블로크자료의 읽기쓰기요청은 표준핵심부루틴(routine)호출에 의하여 buffer_head의 형태로 장치구동기에 주어 진다. 여기에는 블로크장치구동기가 요구하는 모든 정보가 주어 져 있다. 장치식별자는 장치를 유일적으로 식별하며 블로크번호는 어느 블로크를 읽어야 하는가를 구동기에 알려 준다.

모든 블로크장치들은 같은 크기를 가진 블로크들의 선형모임으로 볼수 있다. 물리적블로크장치에 대한 접근속도를 높이기 위하여 Linux는 블로크화된 캐쉬를 리용한다. 체계에서 모든 블로크화된 캐쉬들은 새로운 완충기나 이미 리용된 완충기에 관계없이 이 캐쉬의 어데인가에 보존된다. 이 캐쉬는 모든 물리적블로크장치들에 공유되는데 어느 한 시각에 캐쉬에는 여러개의 블로크완충기가 있게 된다. 이 완충기들은 체계의 블로크장치들중의 어느 하나에 속하며 흔히 여러가지 상태에 있게 된다. 만일 캐쉬로부터 유효한 자료를 제공 받을수 있으면 체계는 물리적장치에 대한 접근을 기억하게 된다.

블로크장치로부터 자료를 읽거나 혹은 거기에 자료를 쓰는데 리용되는 블로크완충기는 캐쉬로 들어 간다. 일정한 시간이 초과되면 완충기는 보다 새로운 봉사를 기다리는 완충기를 위하여 캐쉬로부터 제거될수도 있고 혹은 자주 접근되는 완충기라면 캐쉬에 그냥 남겨 둘수도 있다. 캐쉬안에 있는 블로크화된 완충기들은 자기의 블로크식별자와 완충기의 블로크번호에 의하여 유일하게 식별된다.

캐쉬는 2개의 기능부분으로 구성된다.

하나는 블로크화된 자유완충기(free block buffer)의 목록인데 지원되는 완충기크기에 해당하는 한개 목록이 존재하며 체계의 블로크화된 완충기들은 이 목록에 대하여 먼저 생성된것이 먼저 제거되는 대기렬로 된다. 현재 지원되는 완충기크기는 512, 1024, 2048, 4096, 8192byte들이다.

다른 하나는 캐쉬 그자체이다. 이 부분은 꼭같은 하쉬첨수를 가지는 완충기사슬에 대한 지적자들의 벡토르로 구성되는 하쉬표이다.

하쉬첨수는 자체의 장치식별자와 자료블로크의 블로크번호로부터 생성된다.

그림 3-4는 몇개 입구점을 가진 하쉬표를 보여 준다. 블로크화된 완충기들은 자유목록중의 어느 하나에 있든가 혹은 캐쉬에 있게 된다. 블로크화된 완충기들이 캐쉬에 있을
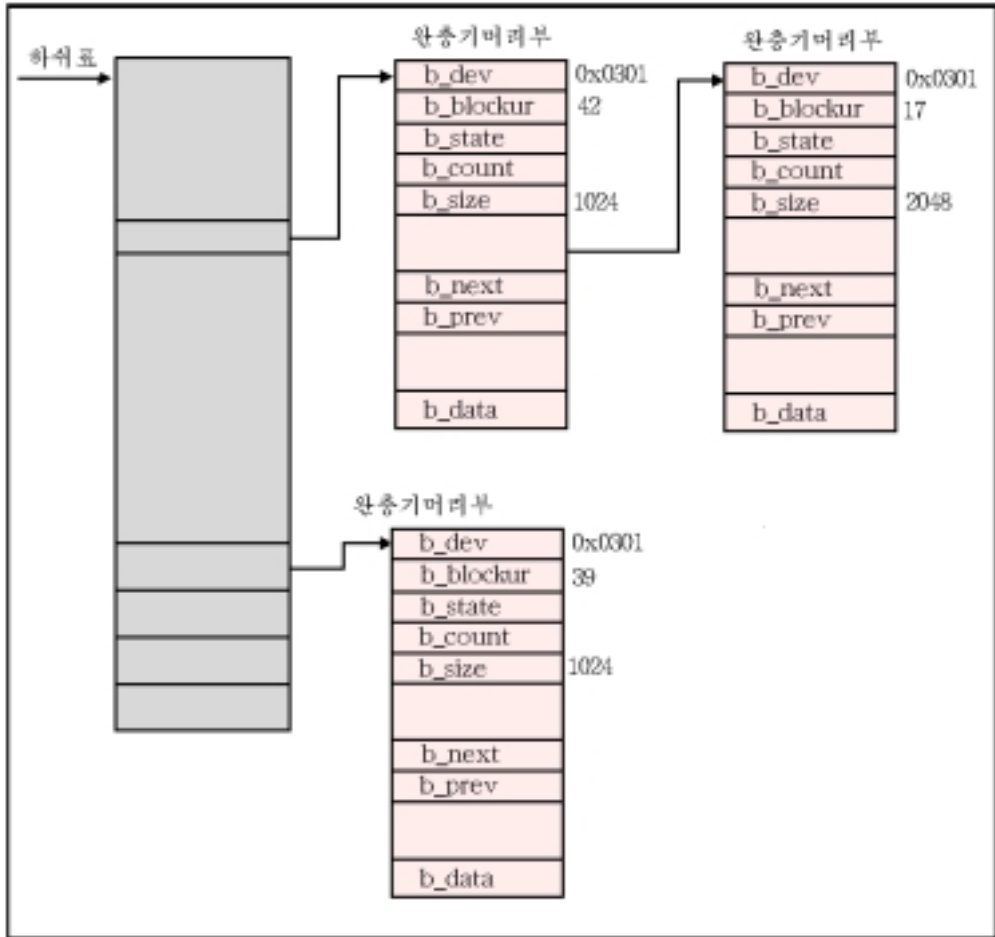
그림 3-4. 캐쉬

때도 역시 가장 최근에 사용된 목록(LRU)에서 대기렬을 형성한다.

매개 완충기형에는 하나의 LRU가 존재하며 이것들은 체계가 어느 한 형의 완충기들에 대하여 작업을 진행하는데 리용된다.

레를 들어 새 자료를 디스크의 비여 있는 완충기에 써넣는 경우를 들수 있다.

완충기의 형은 그것의 상태를 반영하는데 현재 Linux는 다음의 형들을 지원한다.

clean	사용되지 않은 새 완충기
locked	기록을 기다리면서 잠그어 져 있는 완충기
dirty	새롭고 유효한 자료를 포함하며 기록될수는 있으나 아직까지 쓰기문서가 정해 지지 않았다.
shared	공유된 완충기
Unshared	한번 공유되였으나 현재는 공유되지 않은 완충기

파일체계가 완충기의 읽기를 요구할 때마다 그 기초를 이루는 물리적장치들은 캐쉬로부터 블로크를 선택하려고 한다. 만일 캐쉬로부터 완충기를 선택할수 없으면 적합한 크기의 자유목록으로부터 지워 진 완충기를 선택하며 이 새 완충기가 캐쉬로 가게 된다. 만일 요구한 완충기가 캐쉬에 있으면 그 완충기는 새롭게 갱신된것일수도 있고 갱신되지 않은것일수도 있다.

갱신되지 않은것이거나 새 블로크화된 완충기라면 파일체계는 장치구동기에 디스크로부터 적합한 자료블로크를 읽어 들일것을 요구해야 한다.

모든 캐쉬들과 마찬가지로 캐쉬는 그것이 효과적으로 동작하며 캐쉬를 리용하는 블로크장치들사이에 캐쉬입구점들을 정교하게 배정할수 있도록 유지관리되여야 한다.

Linux는 캐쉬에 대하여 많은 상세한 관리적작업을 수행할수 있도록 bdflush핵심부데몬(daemon)을 리용하는데 캐쉬가 리용되고 있는것으로 하여 어떤 일들이 자동적으로 수행되는것도 있다.

bdflush핵심부데몬

bdflush핵심부(daemon)데몬은 체계가 변경된 완충기를 너무 많이 가지고 있을 때 즉 어떤 시각에 디스크에 자료가 완전히 다 씌여 진 완충기를 많이 가지고 있을 때 거기에 대한 동적응답을 제공하는 간단한 핵심부데몬이다.

이 데몬은 체계기동시작시에 핵심부스레드(kernel thread)로서 출발한다. 혼돈을 피하기 위하여 이것을 “kflushd”라고 부르는데 이 데몬은 체계의 프로쎄스를 보아야 할 때 리용하는 ps명령을 써서 볼수 있다. 대체로 이 데몬은 체계안에서 변경된 완충기의 수가 많아 질 때까지 기다리면서 대기(sleep)상태에 있게 된다. 완충기들이 배정되거나 제거될 때마다 체계안의 변경된 완충기들의 수가 검사된다. 체계안의 전체 완충기수의 퍼센트가 커지면 bdflush가 대기상태에서 벗어 난다. 보통 기정값(default threshold)은 60%이지만 체계가 완충기를 비우고 싶은 경우에는 bdflush가 어느 때나 기동할수 있다. 이 값은 관측할수도 있으며 update명령으로 변경시킬수도 있다.

```
# update_d
bdflush version 1.4
```

0 : 60 변경된 블로크를 검사하기 위한 LRU목록의 최대비
1 : 500 bdflush가 능동화될 때마다 써넣기 위한 변경된 블로크의 최대개수
2 : 64 refill_freelist에 의하여 자유목록에 적재될 새 완충기들의 개수
3 : 256 refill_freelist로 bdflush를 능동화하기 위한 변경된 블로크규정값
4 : 15 자유클라스터(cluster)를 주사하기 위한 캐쉬의 퍼센트값
5 : 3000 flush하기전까지 자료완충기가 낡아 지는 시간
6 : 500 flush하기전까지 비자료(dir.bitmap. 등)완충기가 낡아 지는 시간
7 : 1884 캐쉬적재의 평균상수
8 : 2 LAV비(완충기제거를 위한 턱값결정에 리용)

모든 변경된 완충기들은 자료가 기록되여 낡아 질 때마다 BUF_DIRTY LRU목록과 련결되며 또한 bdflush는 자기가 가지고 있는 디스크의 외부에 적당한 수의 완충기를 써넣으려고 한다. 이 값은 update명령에 의하여 관측될수도 있으며 조종할수도 있는데 기정값(default)은 500이다. 매개의 파일체계가 서로 관련되여 있는 한 캐쉬와 완충기의 배정은 아주 단순하다.

파일체계는 캐쉬로부터 페지캐쉬나 새로운 완충기를 위한 새 페지를 요구하며 캐쉬는 핵심부의 자유페지목록이 이 요구를 만족시키도록 해당한 페지들을 제거한다.

만일 이와 같은 조작을 계속하면 결국 모든 기억에 대해서도 적용되는 결과가 초래될것이다. 이것을 피하기 위하여 Linux의 설계자들은 분리기술 즉 다음과 같은 2중재생방법을 제안하였다.

첫째 만일 핵심부프로쎄스가 자유페지를 요청할 때 요청이 절박하지 않으면(즉 조종용의 긴급한 중단이 없으면) 기억배정자에 대한 호출이 기억을 재생시키기 위한 기능을 얻기 위하여 기억재생기능을 차례로 호출한다.

둘째 만일 기억배정호출이 발생하지 않으면 우선 순위가 낮은 데몬인 kswapd가 정규적으로 자유기억상태를 검열하고 더 낮은 경우에는 기억재생을 시작한다.

매 경우에 기억재생기능은 동일하다. 핵심부는 파일을 가상기억, 페지, 캐쉬에 대응시키는 과정을 처리하는 페지표를 포함하는 여러가지 자료구조를 거치면서 순환한다. 핵심부는 해당 페지를 통과한 마지막시각부터 접근되지 않은 페지를 찾고 만일 그 페지를 찾으면 리용하지 않고 자유목록에 그것을 돌려 보낸다.

Kswapd

kswapd는 페지들을 주기적으로 순환하면서 유연한 동작을 보장할수 있도록 하는데 충분한 자유페지를 확보하게 한다.

/proc/sys/vm/ kswapd에는 3개의 파라메터들이 있다.

```
/proc/sys/vm/ $ cat kswapd
512   32    32
```

첫번째 파라메터는 tries_base이다. 이 파라메터는 kswapd가 매 통과마다 비우려고 하는 4~8개로 분할된 페지수이다. 여기서 보다 큰 파라메터는 전체적인 페지교환(swap)성능을 제고할수 있도록 교환페지공간을 더 빨리 해제시킨다.

두번째 파라메터는 tries_min인데 kswapd가 매 통과에서 비워야 할 페지의 최소개수를 나타낸다.

세번째 파라메터는 kswapd가 매 통과에서 써넣은 페지의 수이다. 값이 크면 kswapd가 한개 I/O에서 더 많은 페지를 가지게 될수 있으며 따라서 성능이 제고된다. 그러나 지나치게 값이 크면 매우 큰 I/O조작이 발생하여 정상동작이 페쇄될수 있다.

파일체계객체

Linux VFS는 블로크장치우에서 자료를 기억, 접근, 관리하기 위한 객체(object)들의 모임과 호상작용한다. 이러한 객체들은 다음과 같다.

▼ 파일(file). 파일은 자료를 읽거나 혹은 쓸수 있고 또한 기억에 넘겨 질수도 있으며 때로는 파일이름목록을 파일로부터 읽을수도 있다. 파일은 UNIX가 가지고 있는 파일서술자(file descriptor)개념에 아주 밀접히 대응한다. Linux에서 파일은 struct. file−operation에 기억된 몇개의 방식을 가진 struct. file에 의하여 서술된다.

- 색인마디(Inodes). 색인마디는 파일체계안에 있는 기본객체를 말한다. 색인마디는 정규파일일수도 있고 등록부일수도 있으며 기호적련결이나 어떤 다른 객체일수도 있다. VFS는 서로 다른 객체들사이에 뚜렷한 차이는 두지 않지만 실제적으로 파일체계를 실현하는데 적당한 성질을 부여하여 보다 높은 준위의 핵심부가 서로 다른 객체를 취급할수 있게 일정한 차이를 두도록 한다. 매개의 색인마디는 struct. Inode_operation에 기억되여 있는 여러가지 방식을 가지는 struct inode에 의하여 표현된다. 파일과 색인마디는 거의 근사하지만 일정한 차이를 가진다.
  우선 색인마디가 가지고 있는 어떤 내용들이 파일에는 전혀 없는것이다. 이에 대한 좋은 실례는 기호적련결이다. 반대로 색인마디를 가지지 않는 파일도 있는데 특히 pipe(pipe는 이름은 아니지만)와 소케트(unix의 구역소케트는 아니다.)를 들수 있다. 또한 파일은 색인마디가 가지고 있지 않는 상태정보도 가지고 있다. 여기서는 특히 position을 들수 있는데 이것은 파일에서 다음것의 읽기나 쓰기를 진행해야 할 위치를 지적한다.
- 파일체계(File System). 파일체계는 뿌리(root)라고 하는 한개의 특징적인 색인마디를 가지는 색인마디들의 집합이다. 다른 색인마디들은 뿌리로부터 출발하여 접근할수도 있으며 파일이름을 조사하여 접근할수도 있다. 파일체계는 체계안에 있는 모든 색인마디들에 대하여 평등하게 적용하는 여러가지 특성을 가지고 있다. 이 특성들중에서 어떤것은 READ_ONLY flag와 같은 기발이다. 또 다른 중요한 특성의 하나는 blocksize이다. 매개 파일체계는 struct super_block에 의하여 표현되며 이것은 struct super−operations에 기억된 몇가지조작을 가지고 있다. Linux에는 상위블로크와 장치번호사이에 강한 상관관계가 존재한다. 매개 파일체계는 파일체계가 존재하는 유일한 장치를 가져야 한다. 어떤 파일체계(nfs나 proc)는 실제 장치에 필요 없는것처럼 만들어 진것도 있다. 이때 이름이 밝혀 지지 않은 장치 major에 대하여 번호 0이 자동적으로 붙여 진다. Linux에서 매 파일체계의 형은 structfile_system_type에 의하여 표현된다. 이것은 곧 한가지 조작 즉 read_super를 포함하는데 이것은 super_block가 주어 진 파일체계를 표현한다는것을 실증해 준다.

▲ 이름(name). 파일체계안의 모든 색인마디는 이름에 의하여 접근된다. 일련의 파일체계들에서는 이름 대 색인마디검색처리가 값 눅게 실현되기때문에 LinuxVFS층은 현재 능동상태의 캐쉬와 가장 최근에 리용된 이름을 보존한다. 이러한 캐쉬가 바로 디캐쉬(dcache)이다. 디캐쉬는 기억내에 나무형태로 만들어

진다. 나무안에서 매개 마디는 주어 진 이름을 가지는 등록부안의 색인마디에 대응한다. 나무에서 색인마디는 한개이상의 마디와 관련될수 있다. 디캐쉬는 파일나무의 완전한 복사가 아니며 해당 나무의 고유한 접두사이다. 이것은 파일나무의 임의의 마디가 캐쉬안에 있으면 그 매듭의 매 상위도 역시 그 캐쉬안에 있다는것을 의미한다. 나무에서 매개 마디는 struct.dentry_operations에 기억된 여러개의 방식을 가지는 struct dentry에 의하여 서술된다. 덴트리(dentry : d입구점)들은 파일과 색인마디사이에서 중간요소로서 작용한다. 매개 파일은 열려져 있는 덴트리를 지적하며 매개 덴트리는 그것이 참조하는 색인마디를 지적한다. 이것은 매개 열려 진 파일에 대하여 그 파일의 덴트리와 그우의 모든 상위들이 캐쉬에 기억된다는것을 암시해 준다.

다음절에서는 이 객체들의 구체적인 내용과 Linux VFS가 이 객체들과 어떻게 호상작용하는가에 대하여 고찰한다.

파 일

파일구조는 linux/fs.h에서 다음과 같이 정의된다.

```
struct fown_struct {
    int pid ;            /* SIGIO가 전송되여야 할 pid 혹은 -pgrp */
    uid_t uid, euid;     /* 소유자를 설정하는 프로쎄스의 uid/euid */
    int signum ;         /* IO에 넘겨 질 posix.1b rt 신호   */
} ;
struct file {
    struct list_head         f_list;
    struct dentry            *f_dentry;
    struct file_operations   *f_op;
    atomic_t                 f_count;
    unsigned int             f_flags;
    mode_t                   f_mode;
    loff_t                   f_pos;
    unsigned long f_reada,   f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct       f_owner;
    unsigned int             f_uid, f_gid;
    int                      f_error;
    unsigned long            f_version;
    /*need for tty driver,and maybe others*/
    void                     *private_data;
};
```

마당들의 의미는 다음절에서 설명한다.

f_list

이 마당은 파일을 여러개 목록들중에서 어느 하나와 서로 련결한다. 매개 능동파일체계에는 상위블로크안의 s_files지적자로 시작되는 한개의 목록이 존재한다. 또한 매개 자유나무구조(fs/file_table.c의 free_list)에 대해서도 하나가 존재한다. 그리고 pipe와 같은(fs/file_table.c의 anon_list) 이름이 밝혀 지지 않은 파일에 대해서도 하나가 존재한다.

f_dentry

이 마당은 주어 진 파일에 대한 색인마디를 지적하는 디캐쉬입구점을 기록한다. 만일 색인마디가 정규파일체계에서는 존재하지 않는 pipe와 같은 객체를 참조하면 덴트리는 d_alloc_root로 생성된 뿌리덴트리이다.

f_op

이 마당은 해당 파일에서 리용해야 할 조작방식을 지적한다.

f_count

해당 파일에 대한 참조회수이다. 매개의 서로 다른 사용자프로쎄스파일서술자에 대하여 한개가 존재하며 매 내부리용에 대해서는 +1인 값을 가진다.

f_flags

이 마당은 읽기/쓰기, 비블로크화, 전용추가와 같은 접근형식을 주어 진 파일에 대하여 기발로 기억한다. 이것들은 해당 구성방식포함파일 asm/fcntl.n에서 정의한다. 이 기발들중의 일부는 열기시간에만 관계되며 f_flags에 기억되지 않는다. 이 제외된 기발들은 O_CREAT, O_EXCL, O_NOCTTY,O_TRUNC이다.

이 목록은 fs/open.c의 flip_open에 있다.

f_mode

F_flags의 아래쪽에 있는 두개의 비트들은 개별적인 읽기나 쓰기접근정보를 이끌어내기 어려운 경우에 읽기나 쓰기접근을 부호화한다.

f_pos

f_pos 는 현행파일의 위치를 기록하며 그 위치는 파일이 O_APPEND기발을 가지지 않을 때 다음읽기요청과 다음쓰기요청에 리용된다.

f_reada, f_remax, f_raend, f_ralen, f_rawin

이 5개의 마당은 파일상에서 순차적접근패턴을 계속 추적하는데 리용되며 앞으로

얼마나 읽어야 하는가를 결정한다.

앞방향읽기에 대해서 분리부분이 있을수도 있다.

f_owner

이 구조체는 프로쎄스 id와 새롭게 리용되는 자료와 같은 파일에 어떤 사건이 생길 때 프로쎄스에 전송할 신호를 기억한다.

현재 건반, 마이크, 직렬포구 그리고 망소케트들은 이 특성(kill_fasync을 통하여)을 리용하는 파일로 볼수 있다.

f-uid, f_gid

이 마당들은 소유자에 대한 설정과 파일을 연 프로쎄스그룹에 대한 설정을 얻는다.

이 설정은 전혀 사용되지 않는것처럼 보인다.

f_error

f_error는 NFS클라이언트파일체계가 쓰기오유를 귀환시키기 위하여 리용하는 마당이다.

fs/nfs/write.c에서 설정되여 fs/nfs/file.c에서 검사되며 mm/filemap. c:gerneric_file_write에서 리용된다.

f_versiob

이 마당은 기본파일체계가 캐쉬상태를 지원하며 무효해 진 캐쉬를 검사하는데 리용되는 마당이다.

파일은 f_pos값을 변화시킬 때마다 같이 변경된다.

실례로 ext인 파일체계는 이 마당을 등록부가 언제 변경되였는가를 검출하기 위하여 색인마디안의 i_version마당과 접속시키는데 리용한다.

private_data

이 마당은 매 열린 파일당 파일정보를 기억시키기 위하여(coda에서 신임장과 같은) 많은 장치구동기들이 리용하는 마당이며 지어는 일부 파일체계들에서도 리용한다.

파일함수

파일함수목록은 linux/fs.h에 다음과 같이 정의되여 있다.

```
typedef int(*filldir_t)(void*, const char*, int, off_t, int_t);

struct file_operations {
loff_t(*llseek)(struct file*, loff_t, int) ;
ssize_t(*read)(struct file*, char*,size_t, loff_t*) ;
int(*readdir)(struct file*, void*,filldir_t) ;
```

```
    unsigned int(*poll)(struct file*,struct poll_table_struct*);
    int(*ioctl)(struct inode*, struct file*, unsigned int,
                  insigned long)
    int(*mmap)(struct file*, struct vm_area_struct*);
    int(*open)(struct inode*, struct file*);
    int(*flush)(struct file*);
    int(*release)(struct inode*, struct file*);
    int(*fsync)(struct file*, struct dentry);
    int(*fasync)(int, struct file*, int);
    int(*check_media_change)(kdev_t dev);
    int(*revalidate)(kdev_t dev);
    int(*lock)(struct file*, int, struct file_lock*);
};
```

llseek

이 함수는 seek체계호출을 실현한다. 만일 정의되지 않으면 fs/read_write.c로부터 default_llseek가 대신 리용된다. 이 함수는 기대하였던것처럼 f_pos를 갱신하며 f_read마당과 f_version마당을 변화시킨다.

read

read는 read체계호출과 실행가능한것들의 적재나 분배파일의 읽기와 같은 다른 경우의 파일읽기를 지원하는데 리용된다. 이 기능은 pread나 pwrite체계호출을 제외하고는 보통 file구조체마당의 f_pos에 대한 지적자의 편위값(offset)(마지막인수)을 갱신한다. 블로크장치우의 체계파일용으로 mm/filemap.c에 있는 generic_file_read루틴이 있다. 이 루틴은 색인마디가 readpage로 정의된 조작을 가지고 있다면 이 조작에 리용할수 있다.

write

이 조작은 write체계호출을 사용할 때와 같이 파일에 자료를 기록하는 조작이다. 이 조작은 자료가 주어 진 장치에 도달했다는것을 필수적으로 담보하지는 못하지만 파일형의 의미에 따라 편리할 때 쓸수 있게 대기렬로 준비시킬수도 있다.

블로크장치의 파일체계에 대하여 이 조작을 실현하기 위하여 generic_file_write를 fs/buffer.c로부터 block_write_partial_page와 결합하여 리용할수도 있다.

readdir

readdir는 등록부라고 가정할수 있는 파일로부터 등록부입구점을 읽고 filldir_t역호출(callback)함수를 리용하여 귀환시키는 기능을 수행한다. 이 함수는 이름에 대한 지적자, 이름의 길이, 이름을 찾는 파일안에서의 위치 그리고 이름과 관련된 색인마디번호와 함께 넘겨 진 void*호출자를 얻는다.

만일 filldir역호출이 령이 아닌 값을 귀환하면 readdir는 충분하다고 가정하고 역시 귀환한다. Readdir가 파일의 끝에 도달하면 0값을 가지고 귀환한다. 달리 말하면 readdir는 어떤 입구점들이 filldir에 주어 진후에야 곧 귀환한다는것을 말한다. 이 경우에는 령 아닌 값이 귀환되여야 한다. 오유가 있는 경우에는 부의 값이 귀환된다.

poll

poll은 select와 poll체계호출을 실현하는데 리용한다. 이 경우에 poll_table_entry를 그것이 넘겨 지는 poll_table_ struct에 추가하여야 한다.

ioctl

ioctl은 ad hoc ioctl기능을 실현한다. 만일 ioctl요청이 알려 진 요청 (FIBMAP, FIGETBSZ, FIONREAD)모임중의 하나가 아니라면 그 요청은 토대파일실현에 넘겨진다.

mmap

이 루틴은 기억에 대한 파일의 넘기기를 실현한다. 이 기능은 흔히 generic_file_mmap를 리용하여 실현한다. 이 과제는 마치 넘기기가 허용된다는것을 립증하여 적당한 객체를 지적하기 위한 vm_area_struct의 vm_ops마당을 설정하려는것처럼 보인다.

open

이 조작은 새로운 파일이 색인마디에서 열려 진 때에 호출된다. 조작은 필요할 때 열기에 관한 설정을 진행할수 있다. 하지만 이 조작은 많은 파일체계에서는 사용하지 못한다. 열기시에 국부적으로 파일을 취하는 례외처리로서 coda가 있다.

flush

flush는 파일서술자가 닫길 때 호출된다. 이 파일에서 열리는 파일서술자도 있을수 있으므로 반드시 최종적인 파일닫기는 아닐수 있으며 중간열기로 된다. 현재 이 조작을 정의하고 있는 파일체계는 NFS클라이언트(client)만이며 이 NFS는 확정되지 않은 임의의 쓰기후 요청을 폐지한다. flush는 close 체계호출을 통하여 오유상태를 반대로 귀환시킬수 있으며 따라서 검사해야 할 오유가 있으면 그것의 사용을 요구할수 있다.

유감스럽게도 flush가 소거를 위한 마지막호출인지 아닌지를 현실적으로 결정할 수단은 없다.

release

release는 파일우에서 마지막핸들(handle)이 닫길 때 호출된다. 이 조작은 필요한 임의의 특별한 해제작업을 수행할수 있다. release는 어떤것에 대해서는 오유상태를 귀환시키지 못하며 따라서 현실에서는 int형보다 void형이여야 한다.

fsync

이 조작은 fsync나 fdatasync체계호출을 수행한다(이것들은 식별가능하다.). 이 조작은 파일에 대한 모든 미확정쓰기가 성과적으로 장치에 도달할 때까지 귀환될수 없다. fsync는 generic_buffer_fdatasync를 리용하여 부분적으로 실현할수도 있는데 이 generic_buffer_fdata는 넘겨 진 색인마디의 모든 페지우의 변경된 완충기들에 쓰기를 완료한다.

fasync

이 조작은 파일의 FIOASYNC기발이 변경될 때 호출된다. int파라메터는 기발에 새로운 값을 설정한다. 현재 이 방법을 쓰는 파일체계는 없다.

check_media_change

이 방법은 기본매체가 변화되였는지 안되였는지를 검사하는 조작이다. 만일 변화되였다면 참값을 귀환시킨다. 파일체계가 올려태우기하려고 할 때 호출된 디스크구동기의 바깥부분만은 read_super에 있게 된다. 만일 이 점에서 참인 값을 귀환하면 장치와 관련된 모든 완충기들은 정확하게 확증되지 않는다.

revalidate

revalidate는 check_media_change에서 서술한바와 같이 매체가 변경되여 완충기들이 무효화된후에 호출된다. 그러므로 check_media_change가 정의되여야만 의미를 가지게 된다. 이 조작을 그것과 완전히 구별되는 조작인 inode:revalidate와 혼돈하지 말아야 한다.

lock

이 조작은 POSIX의 잠금을 여유조종할수 있게 파일봉사를 하는 함수이다. 이 함수는 FLOCK형식의 잠금에는 리용하지 않는다. 이 조작은 파일체계에 의하여 자기에게 고유한 특정의 수법으로 서로 다른 잠금방법이 실현되는 망파일체계에서는 특별히 쓸모가 있다. 잠금을 설정하거나 해제할 때 먼저 이 방법으로 실현하며 다음 표준 POSIX잠금코드를 리용한다. 이 방법으로 잠금을 실현할수 있으나 국부코드가 실패하면 잠금을 해제할수 없게 된다.

한 프로쎄스에 어떤 잠금이 설정되여 있는가 즉 이 조작에 의하여 귀환된 정보들을 알아 보려고 할 때 국부잠금은 검사되지 않는다.

색 인 마 디

색인마디는 임의의 UNIX파일체계의 기본구성블로크이다. 그러므로 이 중요한 구조에 대한 아주 명백한 관점을 가지는것은 특별히 중요하다. 색인마디는 그자체가 다른 객체 즉 파일이나 등록부를 표현할수 있다. 매개 파일에는 색인마디가 있으며 파일은 그것이 존재하는 파일체계와 파일체계상의 색인마디번호에 의하여 유일하게 식

별된다. 매개 색인마디는 여러가지 조작 혹은 함수에 대한 지적자를 포함한다. 이러한 조작들은 자료의 읽기나 쓰기, 파일안에서 변위의 탐색, 새로운 파일을 만들거나 이미 있는 파일의 이름바꾸기 등을 수행할수 있다. 모든 연산이나 조작들은 색인마디우에서 무효하다. 실례로 사용자는 정규적인 파일우에서 이름을 재정의할수 있다. 왜냐하면 UNIX에서 이름의 재정의는 상위등록부에서의 조작이며 파일 그자체에서의 조작은 아니기때문이다. 한편 파일뿐아니라 파이프(pipe)에 관해서도 찾을수 없다. 매개 색인마디는 그것이 포함하는 자료블로크수에 대한 계수값을 포함한다. 실제적인 자료블로크의 수는 배정된 자료블로크와 간접블로크의 합이다. 명령 fsck는 실제적자료블로크수를 계산하고 그것을 색인마디가 제시하는 실제적블로크수와 비교한다. 만일 색인마디가 부정확한 값을 포함하고 있으면 fsck는 그 값을 맞추기 위하여 연산자를 제시한다. 매개 색인마디는 64bit크기의 마당을 가진다. 이 크기는 색인마디와 관련되는 파일안의 자료바이트수이다. 크기마당의 정확성은 크기마당으로부터 색인마디와 관련된 블로크들의 최대값을 계산하고 색인마디가 주는 실제적블로크수와 대비하여 기대되는 블로크계수값을 비교하는 방법으로 대략적으로 검사된다.

매개 색인마디는 다음의 정보를 포함한다.

▼ 마디가 있는 장치
- 파일방식
- 파일형
- 잠금정보
- 파일길이(파일에서 바이트길이)
- 련결계수값(파일에 대한 련결수)
- 소유자의 사용자와 그룹 ID
- 접근특권준위
- 마지막파일접근시간
- 색인마디의 마지막변경시간

▲ 디스크상의 파일블로크주소(파일자료를 포함하는 내용에 대한 지적자)

Linux는 능동상태의 캐쉬와 가장 최근에 리용한 색인마디를 보존한다. 이 색인마디에 접근하는데는 두가지 경로가 있다.

첫번째는 디캐쉬를 통한 방법이다. 디캐쉬에서 매개 덴트리는 색인마디로 간주되며 따라서 캐쉬에 그 색인마디들이 보존된다.

두번째 경로는 색인마디하쉬표를 리용하는 방법이다.

매개 색인마디는 파일체계상위블로크의 주소와 색인마디번호에 기초하여 8bit하쉬표로 만들어 져 있다. 같은 하쉬값을 가지는 색인마디들은 2중련결목록으로 서로 사슬화된다. 하쉬표를 통한 접근은 iget함수를 리용하여 실현되는데 iget함수는 색인마디를 찾을수 있는 파일체계와 nfsd에 의하여서만 호출된다. 색인마디는 디캐쉬에서 찾을수 없다. 색인마디상에서 하쉬법의 기초는 매개 파일체계가 파일을 32bit로 유일적으로 식별할수 있다

는 가정으로 어느 정도 제한하고 있는데 있다.

NFS파일체계에서는 최소한 이것이 문제이다. 여기서는 하쉬에서의 유일한 식별자로 256bit파일핸들을 리용하고 있다. nfsd는 파일핸들-색인마디넘기기함수를 제공하는 파일체계를 가지고 있으므로 보다 편리하다. 이때 넘기기함수는 파일핸들을 가장 정교한 방법으로 해석해야 한다.

시동시에 핵심부는 핵심부객체들을 위한 캐쉬에 기억된 코드를 생성하는 루틴(핵심부기억배정코드)을 호출하여 파일의 접근성에 대한 일반적정보를 보관하는 표를 초기화한다. 핵심부는 파일이 열릴 때까지 파일조종구조인 첫번째 객체를 실제적으로 생성하지 못한다. 이미 언급한 캐쉬에 기억된 특수한 핵심부객체를 바로 색인마디라고 부른다. 색인마디는 핵심부안에 있으면서 파일체계, 망경로조종기 혹은 다른 기억관리체계에 의하여 관리되는 특별한 객체이다. Linux에서는 보통 C언어로 색인마디를 실현하는데 설계자체는 매우 추상화되고 객체지향화되여 있다.

아래의 코드는 색인마디핵심부구조체이다.

```
struct inode {
    struct list_head            i_hash;
    struct list_head            i_list;
    struct list_head            i_dentry;
    unsigned long               i_ino;
    unsigned int                i_count;
    kdev_t                      i_dev;
    umode_t                     i_mode;
    nlink_t                     i_nlink;
    uid_t                       i_uid;
    gid_t                       i_gid;
    kdev_t                      i_rdev;
    off_t                       i_size;
    time_t                      i_atime;
    time_t                      i_mtime;
    time_t                      i_ctime;
    unsigned long               i_blksize;
    unsigned long               i_blocks;
    unsigned long               i_version;
    unsigned long               i_nrpages;
    struct semaphore            i_sem;
    struct inode_operations     *i_op;
    struct super_block          *i_sb;
    wait_queue_head_t           i_wait;
    struct file_lock            *i_flock;
    struct vm_area_struct       *i_mmap;
```

```
        struct page                         *i_pages;
        spinlock_t                          i_shared_lock;
        struct dquot                        *i_dquot[MAXQUOTAS];
        struct pipe_inode_info              *i_pipe;
        unsigned long                       i_state;
        unsigned int                        i_flags;
        unsigned char                       i_sock;
        atomic_t                            i_writecount;
        unsigned int                        i_attr_flags;
        __u32                               i_generation;
    union {
        ....
        struct ext2_inode_info     ext2_i;
        ....
        struct socket           socket_i;
        void                   *generic_ip;
    } u;
};
```

이 구조체에서 가장 중요한 마당들에 대하여 보기로 한다.

i_hash

i_hash련결목록은 같은 하쉬바케트에 하쉬화한 모든 색인마디들을 서로 련결한다. 하쉬값은 상위블로크의 주소와 색인마디의 번호에 기초한다. 이 하쉬값은 특정한 마디의 탐색속도를 비약적으로 높인다.

i-list

i_list련결목록은 여러가지 상태에 있는 색인마디를 련결한다. 여기에는 주어 진 파일체계의 모든 변경된 색인마디들을 보존하는 superblock->s_dirty, 사용되지 않는 색인마디들을 목록화한 inode_unused, 능동사용중에 있는 변경되지 않는 색인마디들을 목록화한 inode_in_use들이 포함된다.

i_dentry

i_dentry련결목록은 색인마디로 간주되는 모든 struct dentry들의 목록이다. 이것들은 덴트리의 d_alias마당과 서로 련결된다.

i_version

i_version마당은 변경된 레코드(record)를 리용하기 위하여 파일체계를 사용한다. 대표적으로 i_version은 사건대역변수의 현행값으로 설정되며 그후에 증가된다. 때때로 파일체계코드는 련관된 파일구조체의 f_version에 대한 i_version의 현행값을 지정한다. 파일구조

체를 계속 사용하는 경우에 색인마디의 값의 변동에 대하여 알려 줄수 있으며 필요하면 파일구조체의 캐쉬에 기억된 자료를 소거할수도 있다.

i_nrpages

이 마당은 i_pages와 련결된 페지의 수를 기록하는데 이때 이 값은 색인마디에 캐쉬에 기억되고 add_page_to_inode_queue에 의하여 추가되며 remove_page_ from_ inode_queue에 의하여 감소된다.

i_sem

이 쎄마포(semaphore)는 색인마디에 변화가 있는가를 감시한다. 색인마디에 대하여 비원자적접근을 시도하는 임의의 코드(대기상태에 있는 두개의 련관된 접근)는 먼저 이 쎄마포를 요구해야 한다. 여기에는 블로크의 배정 및 비배정, 여러가지 방향에 대한 탐색과 같은 내용들이 포함된다.

i_flock

i_flock는 색인마디에 잠금을 요구하는 struct file_lock구조체의 목록을 지적한다.

i_mmap

색인마디의 넘기기를 서술하는 모든 vm_area_struct구조체는 vm_next_share와 vm_pprev_share지적자를 서로 련결하며 이 i_mmap지적자는 목록안의 요소를 지적한다.

i_pages

이 마당은 색인마디를 참조하는 페지에서의 캐쉬안의 모든 페지들의 목록이다. 이것들은 이 기억기안의 next와 prev련결상에서 서로 련결된다.

i_shared_lock

이 회전잠금(spin lock)은 i_mmap목록의 vm_next_share와 vm_prev _share지적자를 감시한다.

i_state

3개의 가능한 색인마디상태비트가 있다. I_DIRTY, I_LOCK, I_FREEING

I_DIRTY	변경된 색인마디는 매 상위블로크 s_dirty목록에 있으며 동기가 요청된 다음순간에 기록된다.
I_LOCK	색인마디들이 생성될 때나 읽기, 쓰기하는동안 잠금이 진행된다.
I_FREEING	참조계수값과 련결계수값이 둘다 령이 될 때 색인마디가 가지는 상태이다. 이 상태는 fat파일체계에서 호출되는 igrab의 리용과 같다. fat는 색인마디로 흥미 있는 작업을 할수 있다.

i_flags

i_flags마당은 상위블로크의 s_flags마당에 대응한다. 많은 기발들은 체계범위에서나 혹은 색인마디마다 설정될수 있다.

MS_NOSUID	setuid/setgid는 이 파일에서 허용되지 않는다.
MS_NODEV	색인마디가 장치전용파일이라면 열수 없다.
MS_NOEXEC	이 파일을 실행할수 없다.
MS_SYNCHRONOUS	모든 쓰기가 동기되여야 한다.
MS_MANDLOCK	강제잠금이 수행된다.
S_QUOTA	Quotas(배정몫)이 초기화된다.
S_APPEND	파일을 덧붙이기만 할수 있다.
S_IMMUTABLE	뿌리에 의하여서도 파일이 변경되지 않는다.
MS_NOATIME	이 파일이 접근될 때 색인마디상에서 접근시간을 갱신하면 안된다.
MS_ODD_RENAME	NFS에 관계된다.

i_writecount

이 마당이 정수이면 쓰기접근을 요구하는 클라이언트(파일이나 기억배치도)의 수를 계수한다. 부수이면 이 수의 절대값이 현재 VM_DENYWRITE넘기기의 수를 계수한다. 한편 0이면 쓰거나 혹은 쓰기로부터 다른것을 정지시키는 일이 없다.

i_attr_flags

이 마당은 전혀 사용되지 않으며 오직 ATTR_FLAG_SYNCRONOUS, ATTR_FLAG_APPEND, ATTR_FLAG_IMMUTABLE, ATTR_FLAG_NOATIME의 조합으로 되는 ext2_real_ inode에 의하여서만 설정된다.

i_generation

i_generation의 목적은 지우기/재리용의 전후에 색인마디사이를 구별할수 있게 하는것이다.

NFS에서는 이것이 아주 중요하다. 현재 ext2와 nfsd에서만 이 마당을 보존한다. 이 마당의 리용을 그렇게 정의하였다고 하여 이것을 VFS계층에 도입할수 있다는것은 아니다. 오히려 매개 파일체계는 주어 진 색인마디에 대하여 유일한 파일행들을 제공할 기회가 있어야 하며 그래야 매개는 가장 좋은 유일성을 담보할수 있다.

색인마디에 작용하는 함수

다른 객체와 마찬가지로 색인마디는 그것들우에서의 표준함수들의 모임에 의하여 조종된다(조작이라고도 한다.).

색인마디의 조작구조체는 다음과 같다.

```
struct inode_operations{
   struct file_operations*default_file_ops;
   int(*create)(struct inode*, struct dentry*,int);
   struct dentry*(*lookup)(struct inode*,struct dentry*);
   int(*link)(struct dentry*, struct inode*,struct dentry*);
   int(*unlink)(struct inode*, struct dentry*);
   int(*symlink)(struct inode*, struct dentry*, const char*);
   int(*mkdir)(struct inode*, struct dentry*, int);
   int(*rmdir)(struct inode*, struct dentry*);
   int(*mknod)(struct inode*, struct dentry*, int, int);
   int(*rename)(struct inode*, struct dentry*, struct inode*,
            struct dentry*);
   int(*readlink)(struct dentry*, char*, int);
   struct dentry*(*follow_link)(struct dentry*, struct dentry*,
            unsigned int);
   int(*get_block)(struct inode*,,long,struct buffer_head*,
            int);
   int(*readpage)(struct file*, struct page*);
   int(*writepage)(struct file*, struct page*);
   int(*flushpage)(struct inode*, struct page*, unsigned long);
   void(*truncate)(struct inode*);
   int(*permission)(struct inode*, int);
   int(*smap)(struct inode*, int);
   int(*revalidate)(struct dentry*);
};
```

default_file_ops

이 마당은 색인마디에서 열린 파일에 대한 기정의 조작표를 지적한다.

파일이 열릴 때 파일구조체안의 f_op마당은 이 표로부터 초기화되며 다음 file_operation표의 open조작이 호출된다. 이 조작은 f_op를 다른(비기정표) 조작표로 바꾸도록 선택할수 있다.

실례로 장치전용파일을 열 때 이 조작을 수행할수 있다.

Create

이것과 함께 다음에 보게 될 8개의 조작들은 등록부색인마디에서만 의미를 가진다.

Create는 VFS가 주어 진 등록부안에서 주어 진 이름으로 파일을 만들려고 할 때 호출된다. VFS와 이름이 존재하지 않는다는것을 검사하면 넘겨 진 덴트리는 색인마디지적자가 NULL이라는것을 의미하는 부의 덴트리로 된다. 만일 성공하면 create는 get_empty_inode로 캐쉬로부터 새로운 자유색인마디를 생성하며 마당을 채워 넣고 그것을

insert_inode_hash를 리용하여 하쉬표에 삽입한다. 다음 mark_inode_dirty를 리용하여 하쉬표에 삽입한다. 다음 mark_inode_dirty를 리용하여 그것이 변경된것이라는 표식을 달고 d_instantiate에 의하여 디캐쉬에 서술한다.

int변수는 파일의 방식(mode)을 표현하는데 이를 리용하여 방식이 S_IFREG라는것을 지적하며 요구되는 허용비트를 정의한다.

lookup

lookup는 이름이(덴트리에 의하여 주어 진) 등록부(inode에 의하여 주어 진)안에 존재하는가를 찾아 보며 있는 경우에 d_add를 리용하여 덴트리를 갱신한다. 이 과정은 색인마디의 찾기와 적재를 포함한다.

만일 찾다가 실패하면 색인마디지적자값이 NULL인 부의 덴트리를 귀환시켜 이 내용을 알려 준다. 또한 오유나 NULL을 귀환시킬뿐아니라 덴트리를 귀환시킬수도 있는데 이 경우에 넘겨 진 덴트리는 해제된다. 이러한 가능성이 실제적으로 리용되는가는 아직 불투명하다.

link

link조작은 첫번째 덴트리에 의하여 참조된 이름으로부터 두번째 덴트리에 의하여 참조된 이름까지의 공고한 련결을 실현한다. 이때 두번째 덴트리는 색인마디에 의하여 참조되는 등록부안에 있다.

만일 성공하면 새로운 덴트리(부의 덴트리였던)에 련결파일의 색인마디를 접수하기 위하여 d_instantiate를 호출한다.

unlink

unlink는 색인마디가 참조하는 등록부로부터 덴트리에 의하여 참조된 이름을 삭제한다. 성공하면 d_delete가 호출된다.

symlink

이 마당은 주어 진 값을 가지는 주어 진 이름으로 주어 진 등록부에 기호이름을 생성한다. 성공하면 덴트리에 새 색인마디를 련결하기 위하여 d_instantiate를 호출한다.

mkdir

주어 진 상위, 이름, 방식을 가지는 등록부를 생성한다.

rmdir

이름으로 지적된 등록부(비여 있는 경우)를 지우며 덴트리 d_delete로 소거한다.

mknod

주어 진 상위, 이름, 방식 그리고 장치번호를 가진 장치전용파일을 생성한다. 다음

덴트리에 새 색인마디련결 d_instantiate을 호출한다.

rename

첫 색인마디와 입구점이 등록부와 현재 존재하는 이름을 참조한다. rename은 두번째 색인마디와 덴트리에 의하여 주어 진 이름과 상위를 가지도록 객체의 이름을 재정의한다. 새로운 상위가 변경된 이름의 하위가 아니라는것을 포함하여 모든 일반적검사가 완료된다.

readlink

덴트리에 의하여 참조되는 기호적련결을 읽어 낼수 있으며 이때 값은 int로 주어 진 최대길이로 사용자완충기(copy_to_user)에 복사된다.

follow_link

어떤 등록부(첫번째 덴트리)와 다른 등록부(두번째 덴트리)안에 이름이 있으면 등록부로부터 이름을 동반하는 명백한 결과는 두번째 덴트리에서 찾을수 있다.

만일 색인마디가 어떤 다른 불투명한 객체를 요구하면 결과는 기호적련결에서와 마찬가지로 적당한 새로운 덴트리를 귀환시키도록 follow_link를 제공한다. int인수는 namei 검색에 대한 부분에서 설명한 lookup기발의 수를 포함한다.

get_block

이 조작은 지정된 파일의 블로크를 보관하는 장치블로크를 찾는데 리용된다. inode와 long은 관측되고 있는 파일과 블로크수(블로크수는 파일체계의 블로크크기에 의하여 분할된 파일변위이다.)를 지적한다. get_block는 buffer_head의 b_dev와 b_blocknr마당을 초기화하며 때로 b_state기발로 변경시킨다. int인수가 령이 아닐 때 새 블로크가 존재하지 않으면 새 블로크가 배정된다.

readpage

readpage는 mm/filemap.c에 의해서만 호출된다.

호출방식

▼ generic_file_readahead와 filemap_nopage로부터 try_to_read_ahead
- do_generic_file_read
- sys_sendfile
- filemap_nopage

▲ generic_filemap는 null이 아닐것을 요구

따라서 이 조작은 sendfile체계호출을 리용하기 위하여서나 혹은 generic_read_file을 file:read조작에 리용하려고 할 때 파일의 기억에로의(기대했던것처럼) 넘기기에 필요하다. readpage는 페지안에서 실제적인 읽기를 기대하지 못한다. 페지는 읽기를 위하여 반드

시 배렬되여야 한다. 클라이언트들은 자료를 리용하기전에 페지가 잠금해제되기를 기다린다.

readpage는 fs/buffer.c에 정의된 block_read_full_page를 리용하여 실현할수 있다. 이 루틴은 inode:get_block가 정의되여 있고 문제의 블로크에 접근하기 위하여 buffer_heads를 설정한다고 가정한다. buffer_heads는 "end_buffer_io_sys"를 호출하기 위하여 완성의 목적으로 설정되며 페지상의 모든 완충기들이 완료될 때 페지의 잠금을 해제한다.

writepage

writepage는 linux/mm/filemap.c로부터 호출된다. 또한 filemap_write_page, filemap_swapout, filemap_sync_pte 그리고 gurenic_file_mmap로부터 do_write_page에 의하여서도 호출된다. Writepage는 fs/buffer.c로부터 block_write_full_page를 리용하여 실행될수 있다.

block_write_full_page는 block_read_fullpage와 가까운 쌍둥이이다.

이것들의 중요한 차이점은 다음과 같다.

▼ block_read_fullpage는 ll_rw_block로서 읽기를 시작하며 한편 block_ write_fullpage는 완충기들만을 설정하고 쓰기는 시작하지 않는다.
- block_read_fullpage는 령으로 설정된 생성기발들을 가지고 inode:get_block를 호출하며 한편 block_write_fullpage는 그것을 1로 설정한다.

▲ block_read_fullpage는 완료목적으로 호출된 end_buffer_io_async를 얻기 위하여 init_buffer를 호출한다.

이 두개의 루틴들은 류사성과 차이점을 더 잘 정의할수 있도록 비트를 지울수도 있다.

flushpage

flushpage는 mm/filemap.c와 mm/swap_state.c로부터 호출된다.

mm/filemap.c는 페지가 해제되기전에 페지상에서 확정되지 않은 I/O가 없다는것을 확인하기 위하여 truncate_inode_page에 의하여 호출된다.

파 일 체 계

우리가 사용하는 파일체계라는 용어에 일정한 애매성이 있다는것을 고려하면서 이 부분에 대하여 학습하기로 한다.

파일체계라는 말은 ext2, nfs 혹은 coda와 같은 파일체계의 특정한 형이나 부류를 의미하는데 리용될수도 있고 /user, /home이나 혹은 /dev/hda4파일체계 등과 같은 파일체계의 특별한 실례를 의미하는데도 리용될수 있다.

첫번째 경우는 파일체계의 등록을 의미하며 두번째 경우는 파일체계의 올려태우기를 의미한다.

그런데 현재 많은 사람들이 이러한 애매한 용어를 계속 사용하고 있다. Linux는

register_filesystem를 호출하여 새로운 파일체계형에 대하여 알아 본다(부분체계 unregister_filesystem을 호출하는 방법으로 파일체계형을 알지 않고 할수도 있다.).
형식적쓰임은 다음과 같다.

```
#include <linux/fs.h>
int register_filesystem(struct file_system_type*fs);
int unregister_filesystem(struct file_system_type *fs);
```

함수 register_file system은 성공하는 경우 0을 귀환하며 fs=NULL이면 EINVAL을 귀환한다. 또한 fs→NEXT !=NULL 혹은 이미 같은 이름으로 파일체계가 등록되여 있으면 EBUSY를 귀환시킨다.

이 함수는 또한 모듈로서 적재되고 있는 파일체계를 위하여 init_module로부터 혹은 fs/filesystems.c의 filesystem_setup으로부터(직접적으로 혹은 간접적으로)도 호출된다. 함수 unregister_filesystem은 오직 모듈의 cleanup_module루틴으로부터만 호출된다.

이 함수는 성공하는 경우 0을 귀환하며 인수가 등록된 파일체계에 대한 지적자가 아니면 EINVAL을 귀환시킨다(특히 unregister_filesystem(NULL)은 Oops일수 있다.).

파일체계등록과 비등록실례는 fs/ext2/super.c에서 볼수 있다.

```
static struct file_system_type ext2_fs_type={
    “ext2”,
      FS_REQUIRES_DEV/*|FS_IBASKET */,/*ibaskets have unresolved
      bugs */ ext2_read_super,
      NULL
};
int_init init_ext2_fs(void)
{
      return register_filesystem(&ext2_fs_type);
}
# ifdef MODULE
EXPORT_NO_SYMBOLS;
int init_module(void)
{
       return init_ext2_fs();
}
void cleanup_module(void)
{
       unregister_filesystem(&ext2_fs_type);
}
# endif
```

structfile_system_type는 linuxfs.h에서 정의되며 다음의 형식을 취한다.

```
  struct file_system_type{
    const char*name;
    int fs_flags;
    struct super_block*(*read_super)(struct super_block*,
             void*, int);
    struct file_system_type*next;
};
```

name

이름마당은 단순히 ext2, iso9660 혹은 msdos와 같은 파일체계형에 대한 이름을 준다. 이 마당은 열쇠로 사용되며 이미 사용중에 있는 이름으로 파일체계를 등록할수 없다. 이름마당은 또한 현재 핵심부에 등록된 모든 파일체계형을 목록화한 /proc/filesystems파일에 리용될수도 있다. 파일체계가 모듈로서 실현될 때 이름은 모듈의 주소공간을 지적하는데 이 주소공간(vmallocd구역으로 넘기기한)은 사용자가 만일 cleanup_module의 unregister_filesystem으로 등록을 루락시키려고 한다거나 또 cat/proc/filesystems/을 실행하려고 하면 참조되지 않는 이름 즉 개발의 첫 단계에서 파일체계서술자들에 의하여 형성된 공통오유에 대하여 Oops를 참조하게 될것이다.

fs_flags

파일체계의 특성을 기록하는 ad hoc기발의 수이다. 파일체계기발 fs_flags에 대하여 몇가지 설명하기 위하여 가능한 기발들을 소개한다.

FS_REQUIRES_DEV 우에서 언급한대로 올려태우기된 매 파일체계는 어떤 장치 혹은 어떤 장치번호와 련결된다. 만일 파일체계의 형이 FS_REQUIRES_DEV이면 실지 장치는 파일체계가 올려태우기될 때 주어 져야 하며 그렇지 않는 경우 이름이 밝혀 지지 않은 장치가 배정된다. nfs와 procfs는 장치를 요구하지 않는 파일체계에 대한 실례이며 ext2와 msdos는 요구하는 실례이다.

FS_NO_DCACHE 이 기발은 선언은 하지만 전혀 리용되지 않는다. fs.h의 해설문으로부터 알수 있는바와 같이 기본취지는 디캐쉬가 오직 실제적으로 사용중에 있는 파일에 대한 입구점만을 보존하게끔 파일체계를 표식화하는것이다.

FS_NO_PRELIM FS_NO_DCACHE와 같이 이 기발은 전혀 사용되지 않는다. 그 취지는 디캐쉬가 사용중에 있거나 사용된 입구점들을 가지지만 그밖에는 그 어떤것도 캐쉬에 기억되지 않는다는것이다.

FS_IBASKET 다른 vapor기발 : 이 기발은 ibasket에 대하여 취급한 부분을 보기 바란다.

next

next는 모든 file_system_type를 서로 사슬모양으로 련결시키기 위한 지적자이다. 이 지적자는 NULL로 초기화되여야 한다(register_filesystem은 이 지적자를 설정하지 않으며 NULL은 next가 설정되지 않으면 −EBUSY를 귀환시킨다.).

read_super

read_super조작은 파일체계(개체)가 올려태우기될 때 호출된다. struct super_block는 s_dev_나 s_flag마당을 제외하고는 지워 진다(모든 마당이 0).

void*지적자는 mount체계호출로부터 아래로 넘겨 지는 자료에 대하여 지적한다. 꼬리의 int마당은 read_super가 오유에 대해서 통보하겠는지 안하겠는지를 통보한다. 이 마당은 root파일체계를 올려태우기할 때에만 설정된다. Root가 올려태우기될 때 가능한 매 파일체계가 성공될 때까지 차례로 실행된다. 이 경우에 인쇄오유는 말끔히 제거되지 못할수도 있다.

이름 혹은 덴트리

VFS층은 파일의 모든 경로이름을 관리하며 토대파일체계가 그것들을 인식하기전에 디캐쉬입구점들로 변환한다. 이때 기호련결의 과제만은 례외되며 이 과제는 토대파일체계로 그대로 넘겨 진다. 그러면 토대파일체계가 그것을 해석한다. 이 경우 모듈의 경계가 약간 모호해 지는 감이 있을수 있다. 디캐쉬는 많은 struct dentry들로 구성된다. 매 덴트리는 파일체계안의 한개 파일이름성분과 그 이름(이름이 하나면)과 관련된 객체에 대응한다. 매개 덴트리는 dcache.dentry에 존재해야 하며 파일체계 올려태우기관계를 기록해야 하는 그것의 상위를 참조한다. 디캐쉬는 색인마디캐쉬의 기본요소이다. 디캐쉬입구점이 존재하면 색인마디도 역시 색인마디캐쉬에 존재한다. 반대로 색인마디캐쉬에 색인마디가 있을 때 색인마디는 그 캐쉬에 있는 덴트리를 참조한다.

덴트리구조체

덴트리구조체는 linux/dcache.h에서 정의한다.

```
strut qstr {
      const unsigned char * name;
      unsigned int len;
      unsigned int hash;
};
# define DNAME_INLINE_LEN 16
```

```
struct dentry {
      int d_count;
      unsigned int d_flags;
      struct inode * d_inode;    /* where the name belongs to - NULL
                                     s negative* /
      struct dentry * d_parent;   /* parent directory*/
      struct dentry * d_mounts;   /* mount information*/
      struct dentry * d_covers;
      struct list_head d_hash;    /* lookup hash list*/
      struct list_head d_lru ;    /* d_count = 0 LRU list*/
      struct list_head d_child;   /* child of parent list */
      struct list_head d_subdirs; /* our children*/
      struct list_head d_alias;   /* inode alias list*/
      struct qstr d_name;
      unsigned long d_time;       /* used by d_revalidate */
      struct dentry_operations *d_op;
      struct super_block d_sb;     /* The root of the dentry tree*/
      unsigned long d_reftime;     /* last time referenced*/
      void* d_fsdata;              /* fs-specific data*/
      unsigned char d_i name[DNAME_INLINE_LEN]; /*small names*/
};
```

d_count

이 조작은 단순한 참조계수조작이다. 이 계수는 d_subdirs목록을 통한 상위로부터의 참조는 포함하지 않지만 하위로부터 d_parent참조는 포함한다. 이것은 캐쉬의 잎마디만이 0인 d_count값을 가질수 있다는것을 암시해 준다. 이 입구점들은 볼수 있으므로 d_lru에 의하여 서로 련결된다.

d_flags

현재 정의된 파일체계를 실현하여 사용가능한 두개의 기발이 있는데 여기에서는 서술하지 않는다.

이 기발로는 DCACHE_AUTOFS_PENDING과 DCACHE_NFSFS_RENAMED을 들수 있다.

d_inode

해당 이름과 관련된 색인마디에 대한 지적자이다. 이 마당은 NULL일수 있는데 이름이 존재하지 않는다는것을 암시하는 부의 입구점을 가리킨다.

d_parent

이 마당은 상위덴트리에 대한 지적자이다. 파일체계의 뿌리나 혹은 파일에서처럼 닉명의 입구점에 관하여 이 지적자는 포함되는 덴트리자체를 거꾸로 지적한다.

d_mounts

올려태우기된 파일체계를 가지는 등록부에 대하여 d_mount는 그 파일체계의 뿌리덴트리를 지적한다. 다른 덴트리에 대해서는 그자체 덴트리를 거꾸로 지적한다. 올려태우기점에서 파일체계를 올려태우기할수 없으며 따라서 하나이상의 긴 d_mount입구점사슬은 존재할수 없다.

d_covers

이 마당은 d_mount와 반대내용을 표현한다. 올려태우기된 파일체계의 뿌리에 대하여 이 마당은 올려태우기된 등록부의 덴트리를 지적한다. 다른 덴트리에 대해서는 덴트리자체를 지적한다.

d_hash

2중련결목록으로서 한개 하쉬바케트안의 입구점들을 서로 사슬로 결합한다.

d_lru

이 마당은 캐쉬에서 참조되지 않은 잎마디들의 2중련결목록을 제공한다. 목록의 머리부는 dentry_unused의 대역변수이다. 이 변수는 LRU(Least Recently Used : 가장 최근에 사용)에 차례로 기억된다. 핵심부의 다른 부분들이 디캐쉬안의 사용되지 않은 입구점들을 잠금할수 있는 기억이나 색인마디를 개정할것을 요구하면 그것들은 d_lru에서 제거가능한 입구점들을 찾고 그것을 prune_dcache에 의하여 제거할수 있게 준비해 주는 select_dcache를 호출할수 있다.

d_child

이 목록머리부는 해당 덴트리의 d_parent의 모든 하위들을 서로 련결하는데 리용된다. 이에 관해서는 d_sibling이 보다 더 좋은 이름이라고 생각해도 좋다.

d_subdirs

이 덴트리의 모든 하위들을 련결하는 d_child의 머리부이다. 물론 요소들은 파일로 간주해도 좋으며 부분등록부로는 간주하지 않는다. 그러므로 d_child가 더 좋은 이름으로 되며 또 이미 사용되고 있다.

d_alias

파일들은(어떤 다른 파일체계의 객체들) 다중고정련결을 통하여 파일체계안에서 다중이름들을 가질수도 있기때문에 다중덴트리도 같은 색인마디들을 참조할수 있다. 이런

현상이 생길 때 덴트리는 d_alias마당에 련결된다. 색인마디의 i_dentry마당은 이 목록의 머리부로 된다.

d_name

d_name마당은 마당의 하쉬값과 함께 입구점의 이름을 포함한다. 이름부분마당은 덴트리의 d_iname마당을 지적할수 있으며 만약 마당이 충분히 길지 않으면 개별적으로 배정된 문자렬을 지적한다.

d_time

이 마당은 토대파일체계에 의하여서만 리용되는데 그것들이 원하든 원하지 않든지간에 가정적으로 리용될수도 있다. 왜냐하면 이 입구점이 유효성을 다시 검사하여야 한다는것을 강조하는것이 중요하기때문이다.

d_op

d_op는 해당 덴트리를 어떻게 조종하는가에 대한 정의와 함께 struct dentry_operations를 지적한다.

d_sb

이 마당은 객체를 가지고 있는 덴트리에 의하여 참조된 파일체계의 상위블로크를 지적한다.

d_inode->i_sb를 쓰는것보다도 오히려 이것이 필요하겠는가가 명백치 않다.

d_reftime

이 마당은 d_count가 령으로 될 때마다 jiffies에 현재시간을 설정하지만 리용되지는 않는다.

d_fsdata

이 마당은 요구될 때 리용할 파일체계를 정의하는데 쓸수 있다. 이 마당은 현재 파일핸들을 기억하기 위하여 nfs에서만 리용된다(필자는 파일핸들이 색인마디에 해당하고 이름에는 해당되지 않는다고 생각하고 있었지만 여러 nfs봉사자들은 동의하지 않았다.).

d_iname

d_iname은 참조를 쉽게 하기 위하여 이름의 첫 16문자를 기억한다. 만약 이름이 완전히 일치되면 d_name.name이 여기를 지적한다. 그렇지 않으면 따로따로 배정된 기억을 지적한다.

덴트리함수

대다수의 덴트리에 대한 조종은 모든 파일체계에서 공통이며 따라서 덴트리상에서

수행하려고 하는 대부분의 조작들은 dentry_operation목록에 조작방식을 가지고 있지 않는다. 오히려 그것은 몇개의 파일체계를 실현하여 불투명한 방법으로 조종되는 몇가지 조작들을 제공한다. 파일체계는 NULL로서 모든 조작들을 그만두도록 선택할수 있는데 이 경우에는 기정조작이 적용된다.

linclude/linux/dcache.h로부터 구조체정의를 보면 다음과 같다.

```
struct dentry_operations{
    int(*d_revalidate)(struct dentry*, int);
    int(*d_hash)(struct dentry*, struct qstr*);
    int(*d_compare)(struct dentry*,  struct qstr*, struct qstr*);
    void(*d_delete)(struct dentry*);
    void(*d_release)(struct dentry*);
    void(*d_iput)(struct dentry*, struct inode*);
};
```

d_revalidate

이 조작은 경로찾기가 입구점이 아직 유효한가를 알아 내기 위하여 디캐쉬에 있는 입구점을 리용할 때마다 호출된다. 만약 여전히 유효한 경우에는 1을 귀환시키며 그렇지 않은 경우에는 0을 귀환시킨다. 기정값은 귀환값 1로 가정한다. int인수는 이 검색과 관련이 있는 기발을 제공하며 LOOKUP_FOLLOW, LOOKUP_DIRECTORY, LOOKUP_SLAHOK, LOOKUP_CONTINUE중의 임의의것을 포함한다. 이것들에 대해서는 namei에 관한 절에서 서술한다.

이 조작은 공유파일체계에서처럼 VFS층이 어떤 작업을 진행하지 않아도 파일체계가 변경될수 있을 때만 필요하다. 만약 d_revalidate가 0을 귀환하면 VFS층은 디캐쉬로부터 덴트리를 정리하여 간결하게 만든다. 이 작업은 능동상태에서 사용되지 않는 임의의 하위덴트리를 제거하는 d_invalidate에 의하여 수행되며 만일 성공되면 덴트리를 하쉬상태에서 해제한다.

d_hash

만약 파일체계가 유효한 이름이나 혹은 이름과 등가인것들에 대하여 비표준적인 규칙을 가지고 있다면 이 루틴은 유효성을 검사하여 정규하쉬를 귀환시키는 기능을 제공해야 한다.

만약 이름이 유효하면 하쉬가 계산되며(모든 등가적이름들에 대해서도 같아야 한다.) 그것을 qstr인수에 기억한다. 이름이 무효하면 적당한(부의) 오유코드가 귀환되여야 한다.

덴트리인수는 덴트리의 이름이 아직 완성되지 않았으므로 상위이름의 덴트리(qstr에서 찾은)로 된다.

d_compare

이 마당은 두개의 qstr에 대하여 그것들이 서로 같은가를 알아 보기 위하여 비교한다. 두 qstr가 같은 때만 0을 귀환한다. 순서는 중요치 않다.

d_delete

이 마당은 덴트리가 dentry_unused목록에 옮겨 지기전에 참조계수값이 0에 도달할 때 호출된다.

d_release

이 마당은 덴트리가 최종적으로 해방되기 직전에 호출되며 d_fsdata을 해제하는데 리용된다.

d_iput

이 마당은 덴트리가 제거될 때 색인마디를 개방시키기 위하여 iput대신 호출된다. 이 조작은 iput와 등가인 조작들과 그밖의 요구하는 다른 조작들에서도 쓸수 있다.

Linux상위블로크

올려태우기된 매개 파일체계는 super_block구조체에 의하여 서술된다. 파일체계가 올려태우기된다는 사실은 linux/mount.h에서 볼수 있는 선포문인 struct vfsmount에 기억된다는것을 말한다.

```
struct vfsmount
{
   kdev_t mmt_dev;                         /*적용되는 장치*/
   char*mnt_devname;                       /*장치이름 즉 /dev/dsk/hdal*/
   char*mnt_dirname;                       /*올려태우기된 등록부이름*/
   unsigned int mnt_flags;                 /*이 장치의 이름*/
   struct super_block mnt_sb;              /*상위블로크에 대한 지적자*/
   struct quota_mount_options mnt_dquot;  /*디스크배정정의 올려태우기선택*/
   struct vfsmount*mnt_next;               /*련결목록의 다음요소지적자*/
} ;
```

이 vfsmount구조체들은 fs/super.c의 vfsmntlist로부터 출발하는 단순련결목록과 서로 련결된다. 이 목록은 장치 특히 디스크배정코드에 주어 진 올려태우기된 파일체계정보를 찾는데 중요하게 리용된다.

vfsmount가 상위블로크목록과 분리되여 있어야 하는 리유는 상위블로크가 이미 존재하면 read_super파일체계정의조작대신 fs/super.c:get_super()가 fs/super.c:read_super()를 만족

시키기때문이다.

한편 vfsmntlist의 입구점은 파일체계가 올려태우기되지 않으면 곧 련결을 차단한다.

매개 올려태우기는 역시 후에 설명하게 될 디캐쉬에 기록되며 이것은 경로이름을 측정할 때 리용하는 올려태우기정보의 원천으로 된다.

상위블로크구조체

상위블로크구조체에 대하여 간단히 서술한다.

```
struct super_block {
   struct list_head          s_list;  /*첫번째로 유지*/
   kdev_t                    s_dev;
   unsigned long             s_blocksize;
   unsigned char             s_blocksize_bits;
   unsigned char             s_lock;
   unsigned char             s_dirt;
   struct file_system_type   *s_type;
   struct super_operations   *s_op;
   struct dquot_operations   dq_op;
   unsigned long             s_flags;
   unsigned long             s_magic;
   struct dentry             *s_root;
   wait_queue_head_t         s_wait;
   struct inode              *s_ibasket;
   short int                 s_ibasket_count;
   short int                 s_ibasket_max;
   struct list_head          s_dirty;  /*변경된 색인마디*/
   struct list_head          s_files;
   union {
     /* 배치구성된 파일이 여기서 입구점을 얻는다.*/
     void   *generic_sbp;
   }  u;
     /*다음의 마당은 VFS만이 필요*/
     struct semaphore  s_vfs_rename_sem;  /*Kludge*/
};
```

우의 삭제된 부분을 가진 union u의 모든 파일체계정의요소들을 포함하는 완전한 선포문을 보려면 linux/fs.h를 참조할것

s_list

올려태우기된 모든 파일체계의 2중련결목록이다(linux/list.h을 볼것).

s_dev

해당 파일체계가 올려태우기된 장치(알려 지지 않은 이름일수 있다.)

s_blocksize

파일체계의 블로크크기이다. 지금도 리용되고 있는지는 확실치 않다. 2의 루승으로 표시되여야 한다.

s_blocksize_bits

s_blocksize는 2의 루승 즉 log2(s_blocksize)이다.

s_lock

상위블로크의 현재 잠금상태를 지적한다. 이 마당은 lock_super와 anlock_super, lock_kernel에 의하여 관리된다.

s_wait

상위블로크상에서 s_lock잠금을 기다리고 있는 프로세스의 대기렬이다.

s_dirt

상위블로크가 변화될 때마다 설정되고 또 상위블로크가 장치에 씌여 질 때마다 지워지는 기발이다. 이 조작은 파일체계가 올려태우기해제될 때 혹은 sync체계호출에 대한 응답으로 발생한다.

s_type

이 마당은 우에서 설명한 struct file_system_type구조체에 대한 단순한 지적자이다.

s_op

다음에 설명할 struct super_operations에 대한 지적자이다.

dq_op

다음에 설명할 디스크배정조작에 대한 지적자이다.

s_flags

매 색인마디안의 어떤 상태를 결정하기 위한 기발들과 론리합을 취한 기발들의 목록이다.

전체 파일체계에 대해서 적용되는 기발은 하나뿐이며 그것을 여기에서 설명한다. 다

른것들은 색인마디에 대하여 론의한 조건에서 설명한다.

MS_RDONLY 기발이 설정된 파일체계는 읽기전용으로만 올려태우기된다. 쓰기는 허용되지 않으며 간접적인 변경도 불가능하다. 례를 들어 상위블로크에서의 올려태우기시간이나 파일에서의 접근시간 등은 변경할수 없다.

s_magic

이 마당은 장치의 자료가 해당 파일체계에 대응한다는것을 확인하기 위하여 장치로부터 읽어 들이는 식별번호이다. 이 조작은 여러가지 특성을 가진 파일체계들을 서로 구별하기 위하여 Minix에서 리용된것으로 알려 져 있다.

왜 이것이 구조체의 일반적부분으로 되여 있었는가 하는 리유는 명백하지 않으며 그것은 파일체계가 요구하는 파일체계정의부분으로 완전히 제한되지 않는다. 이 마당의 유용성의 하나는 fs/nfsd/vfs.c:nfsd_lookup()에 있는데 proc나 nfs파일체계가 NFS를 통해서는 접근할수 없다는것을 확인하는데 리용한다.

s_root

이 마당은 파일체계의 뿌리로 간주되는 struct dentry이다. 이 마당은 파일체계로부터 뿌리색인마디를 적재하며 그것을 d_alloc_root에 넘겨 주는 방법으로 표준적으로 생성된다. 이 덴트리는 올려태우기명령에 의하여 디캐쉬에 겹쳐 이어 진다(do_mount가 d_mount를 호출).

s_ibasket, s_ibasket_count, s_ibasket_max

이 세가지 마당은 색인마디의 바스케트인데 현재 판본에는 이러한것들이 존재하지 않는다.

s_dirty

i_list마당에 련결된 변경된 색인마디들의 목록이다. 색인마디가 mark_inode_dirty에 의하여 변경된 색인마디로 표식이 붙으면 이 목록에 놓는다. sync.inode가 호출되면 이 목록안의 임의의 색인마디가 파일체계의 write_inode조작에 넘겨 진다.

s_files

해당 파일체계상의 열린 파일들의 목록이다(f_list와 련결된).

실례로 이것은 파일체계가 읽기전용으로 재올려태우기되기전에 쓰기용으로 열린 파일이 있는가를 검사하는데 리용된다.

u.generic_sbp

공용체 u는 매개 파일체계에 대하여 콤파일시에 대략적으로 알려 지는 한개의 파일

체계정의용상위블로크정보구조체를 포함한다. 모듈로서 적재되는 임의의 파일체계는 개별적구조체를 배정해야 하며 u.generic_sbp에 지적자를 배치해야 한다.

s_vfs_rename_sem

이 쎄마포는 등록부의 이름을 재정의하는 동안 파일체계범위의 잠금에 리용된다. 이 조작은 등록부의 이름바꾸기가 그의 하위자체에 대해서는 끝나게 되는 가능한 경쟁조건을 감시하는것으로 표현된다. 이 쎄마포는 등록부가 아닌 객체의 이름바꾸기때에는 필요하지도 않으며 사용되지도 않는다.

상위블로크함수

struct super_operations에 정의된 조작은 다음과 같다.

```
struct super_operations{
   void(*read_inode)(struct inode*);
   void(*write_inode)(struct inode*);
   void(*put_inode)(struct inode*);
   void(*delete_inode)(struct inode*);
   int(*notify_change)(struct dentry*, struct iattr*);
   void(*put_super)(struct super_block*);
   void(*write_super)(struct super_block*);
   int(*statfs)(struct super_block*, struct statfs*, int);
   int(*remount_fs)(struct super_block*, int*, char*);
   void(*clear_inode)(struct inode*);
   void(*umount_begin)(struct super_block*);
};
```

이 조작들은 핵심부의 잠금이 유지된 상태에서만 호출된다. 이것은 블로크의 안전을 보장할수 있다는것을 의미하지만 그것들자체의 병행접근을 막는다는데도 의의가 있다. 모든 조작은 프로쎄스문맥으로부터 호출되며 중단조종자나 bottom half로부터는 호출되지 않는다.

read_inode

이 조작은 올려태우기된 파일체계로부터 정의된 색인마디를 읽기 위하여 호출된다. 이 조작은 fs/inode.c안의 iget에서 나온 get_new_inode로부터만 호출된다. 이 조작에 넘겨 진 struct inode*인수에서 마당 i_sb, i_dev와 특히 i_ino는 색인마디를 어느 파일체계로부터 읽어야 하는가를 지적하기 위하여 초기화된다. 이 조작은 또한 VFS가 주어 진 색인마디에서 조작을 요구하는대로 호출할수 있도록 관련된 struct inode_

operations마당을 지적하기 위하여 struct inode의 i_op마당을 설정하여야 한다(다른것들중에서).

흔히 iget는 그 파일체계에 해당한 색인마디들을 읽기 위하여 특정한 파일체계안에서 호출된다. fs/nfsd/nfsfh.h에는 한가지 주목할만한 례외가 존재하는데 그것은 nfs파일핸들의 정보에 기초한 색인마디를 얻는데 리용된다.

이 조작은 그것을 제공하는 파일체계에 의하여서만(간접적으로) 리용될수 있기때문에 (nfsd를 제외하고) 수출될것을 요구하는것은 부당하다.

이런 경우를 피하는 방법은 단순 32bit색인마디번호보다도 더 좋은 유연성을 가지고 특정한 색인마디를 식별하는것이다.

nfsd의 리용은 파일핸들(혹은 그것의 다른 부분)을 가지며 색인마디를 귀환시키는 대면부로 갱신할수 있다.

write_inode

이 조작은 mark_inode_dirty로 변경된것에 표식을 붙인 색인마디들을 호출한다. 조작은 파일상에서와 파일체계상에서 sync요청이 생길 때 호출된다. 이때 색인마디에서 임의의 정보가 장치적으로 안전하다는것을 확인해야 한다.

put_inode

이 조작은 색인마디상에서 참조계수값이 감소될 때마다 호출된다. 이것을 더 많은 색인마디들이 사용중에 있거나 또 여러명의 사용자를 가진다는것으로 리해하면 안된다는 것을 강조한다.

put_inode는 i_count마당이 감소되기전에 호출되며 따라서 put_inode는 마지막참조인가를 확인하려면 i_count가 1인가 0인가를 검사해야 한다.

이 조작을 정의하고 있는 거의 모든 파일체계들은 색인마디에 대한 마지막참조가 해제될 때 즉 i_count가 1이거나 0일 때는 몇가지 특별한 조종을 진행하는데 리용한다.

delete_inode

delete_inode가 색인마디에 대한 참조계수값이 령으로 될 때마다 호출된다고 정의하면 역시 련결계수값(i_nlink)도 령이라는것을 말해 준다. 파일체계는 이러한 상태를 파일체계의 색인마디를 무효화시키거나 사용된 자원을 해방시키는 상태로 처리한다고 가정할수 있다. 이 조작과 앞의 조작은 i_count마당이 0으로 될 때마다 호출되는 한가지 조작으로 바꿀수 있으며 그러면 파일체계는 그것이 0인 i_nlink로 어떤 특별한 조작을 할수 있는가를 결정할수 있다. 이러한 조작을 현행 파일체계에 의하여 실현시키는데서 유일한 난점은 ext2가 i_count에 무관계하게 put_inode가 호출될 때 ext2_discard_prealloc를 호출하는것이다. 그러면 이것이 더이상 가능하겠는가. 파연 실현할수 있겠는가 하는 문제가 제기된다.

notify_change

이 조작은 색인마디의 속성이 변할 때 즉 새로운 속성모임을 지적하는 인수 struct_iattr가 변할 때 호출된다. 만일 파일체계가 이 조작을 정의하지 않으면 (즉 NULL

이면) VFS는 POSIX표준속성검증을 실현하는 fs/iattr.c:inode_change_ok루틴을 리용한다. 다음 VFS는 색인마디에 낡았다는 표식을 단다. 만일 파일체계가 그자체 notify_change를 수행하면 이 조작은 속성을 설정한 다음 mark_ inode_dirty(inode)를 호출한다. 이 조작을 어떻게 실현하는가 하는 실례는 fs/ext2/inode.c:ext2_notify_change()에서 볼수 있다.

put_super

이 조작은 vfsmntlist로부터 입구점들을 제거하기전에 umount(2)의 마지막단계에서 호출된다. 또한 이 조작은 상위블로크잠금이 유지된 상태에서 호출된다. 전형적인 실현으로는 이 올려태우기개체를 위하여 정의된 파일체계비공개자원 즉 색인마디비트매프, 블로크비트매프, 상위블로크를 포함하는 블로크머리부를 개방하는것과 만일 파일체계가 동적적재모듈로서 실현된다면 계수값을 유지한 모듈을 감소시키는것을 들수 있다. 실례로 fs/bfs/inode.c:bfs_put_ super()을 들수 있는데 아주 간단하다.

```
static void bfs_put_super(struct super_block*s)
{
   brelse(s->su_sbh);
   kfree (s->su_imap);
   kfree(s->su_bmap);
   MOD_DEC_USE_COUNT;
}
```

write_super

VFS가 상위블로크로 하여금 디스크에 쓰기를 요구할것을 결정할 때 호출된다. 호출되는 등록부는 fs/buffer.c:file_fsync, fs/super.c:sync_supers와 fs/super.c:do_umount이다. 읽기전용파일체계에 대해서는 필요없다는것이 명백하다.

statfs

이 조작은 statfs(2)체계호출수행시에 요구되며 실행될 때 fs/open.c : sys_statfs로부터 호출된다. 그렇지 않은 경우 statfs(2)는 실패하며 errno를 ENODEV로 설정한다.

remount_fs

이 조작은 파일체계가 재올려태우기될 때 즉 MS_REMOUNT기발이 mount(2)체계호출에 의하여 설정될 때 호출된다. 또한 파일체계를 재올려태우기하지 않고도 여러가지 올려태우기선택을 변경시킬 경우에 리용할수 있다.

리용상 공통점은 읽기전용파일체계를 쓰기가능한 파일체계로 변화시키는것이다.

clear_inode

이 선택적인 조작은 VFS가 색인마디를 지울 때 호출된다. 이 조작은 특별히 struct.inode의 generic_ip를 리용하는 파일체계에 해당되는 경우로서 색인마디구조체에 동적배정된 자료를 덧붙일 목적을 가진 파일체계가 요구하는 조작이다. 이 조작은 현재 색

인마디에 kalloc로 배정된 자료를 덧붙이는 ntfs와 색인마디번호를 지원하지 못하는 파일체계에서 색인마디번호가 안정하다는 가상적느낌을 주기 위하여 일련의 흥미 있는 조작을 수행하는 fat에 의하여 리용되고 있다.

umount_begin

이 조작은 올려태우기를 해제하기 위하여 MNT_FORCE기발이 주어 지면 올려태우기해제프로쎄스에서 호출된다. 원격봉사기응답과 같은 어떤 외부적사건을 기다리는 정지상태보다는 파일체계상의 어떤 불완전한 거래가 고장을 더 빨리 야기시킨다는데 이 조작의 취지가 있다. umount_begin의 호출은 대체로 능동파일체계를 올려태우기해제가능하게 만들지는 않지만 비종단대기에 있는것보다 오히려 그 파일체계를 리용하는 임의의 파일체계가 파괴될수 있다는데 대하여 강조해 둔다. 현재 NFS는 umount_begin을 제공하는 유일한 파일체계이다.

성능문제와 최량화방책

파일체계성능은 전체적인 체계성능의 주요인자이며 적재를 동반하는 응용프로그람의 특성에 크게 의존한다. 최량성능을 얻기 위하여 토대파일체계배치구성은 응용프로그람의 특성에 맞게 균형이 이루어 져야 한다. 만일 사용자자신이 개발자라면 자기의 응용프로그람이 파일체계를 거쳐 읽기나 쓰기를 진행하는 방법에 대하여 이미 잘 알고 있을것이다. 하지만 만일 자신이 응용프로그람의 관리자라면 파일체계에 주어 지는 I/O의 형태를 리해하기 위하여 응용프로그람을 해석하는데 일정한 시간이 걸릴것이다. 일단 응용프로그람에 대한 리해를 가지기만 하면 토대계층기억장치를 효과적으로 리용할수 있도록 파일체계배치를 최량화할수 있다.

그와 같은 객체들은 다음의것들을 들수 있다.

▼ 토대층장치들에 대한 I/O의 수를 가능한껏 줄이는것
- 가능한껏 작은 I/O들을 더 큰 I/O들로 그룹화하는것
- 디스크찾기를 대기하면서 소비되는 시간을 줄이기 위하여 찾기패턴을 최량화하는것

▲ 실지적인 물리적I/O를 줄이기 위하여 가능한대로 더 많은 자료를 캐쉬에 기억된다는것

거래를 더 빨리 처리하기 위하여 거래의 서로 다른 부분들이 무엇을 할수 있는가를 고찰해야 한다.

거래는 다음과 같은것들로 구성된다.

▼ 거래의 가동일지등록
- 변경되기전에 변경시켜야 할 주제로 되는 자료의 가동일지등록
- 기억으로부터 자료기지 레코드에 접근

- 자료에 의한 여러가지 조작
- 변경이 완료된후 주제가 변경되여 있는 자료의 가동일지등록

▲ 거래완료의 가동일지등록

자료기지만이 아니라 사용자등록파일은 비휘발성기억에 보존되여야 하므로 상당한 I/O수가 요구된다는것을 쉽게 알수 있다.

원시입출력(Raw I/O)

Linux에서 한가지 최신특성은 저준위장치자체로 곧바로 가지 않으며 캐쉬들을 통해서 접근이 조종되지 않는것들중의 하나인 《원시》I/O의 실현이다. 관계형자료기지관리체계(Oracle, Sybase, Informix DB2와 기타 다른 체계)들과 같은 몇가지 응용프로그람들은 자체의 잠금방책을 관리할수 있기때문에 원시장치를 리용하는 쪽에 더 관심을 둔다. 보충적으로 파일체계실행경로를 리해하는것도 체계의 성능을 개선하게 한다.

원시장치들은 정교한 응용프로그람들이 자료의 캐쉬에로의 기억과 일반화된 캐쉬에로의 기억의 휴지시간을 줄일수 있는 완전한 조종을 요구하는 경우에 사용될수 있다. 원시장치들은 또한 체계의 오유사건속에 잃어 진 자료가 하나도 없도록 하기 위하여 디스크에 자료가 즉시에 다 씌여 졌다는것을 확인하는 자료-림계-상태에 리용되기도 한다. 원시장치자원에 관한 초기의 제안들은 매개 블로크장치들을 원시장치마디에 주기 위하여 장치마디의 수를 완전히 중복시켜야 했으므로 적합하지 못한것으로 간주되였다. 많은 상업적UNIX들이 리용하는 방법인 2.4Linux실현에서는 풀장치마디를 리용하는데 이 마디는 어떤 임의의 블로크장치와 련관을 가질수 있다.

이 착상이 더 전진하여 2.4판본에서는 "kiobuf"라는 새로운 객체를 포함하게 되였다. kiobuf는 곧 핵심부페지의 임의의 가지에 대한 추상화된 표현인바 이것은 완충기로서 리용될 핵심부페지안의 자유페지이므로 init.c의 코드에 의하여 기동시간내에 설정된다. 원시I/O는 kiobuf를 생성하고 프로쎄스가 I/O로 리용하고 있는 자료완충기를 포함하는 물리적페지들을 가지는 kiobuf를 거주시키는 형식으로 작업한다. 정확한 물리적기억페지가 자료용으로 처리되기만 하면 kiobuf는 읽기나 쓰기를 위하여 I/O층에 넘겨 진다. I/O층들은 련관된 페지들이 사용자프로쎄스에 속하는가를 알아야 할 필요가 없다. 즉 kiobuf은 그것의 구체적내용들은 내부에 숨기고 있다. 모든 I/O층들에 보이는것은 물리적페지들과 그 페지들에 대한 I/O요청들이다.

프로쎄스자원한계

프로쎄스의 문맥안에는 실행시 프로쎄스들이 리용하는 몇가지 체계자원에 대한 한계가 존재한다. 체계는 이 자원한계에 의하여 조종되는 매 자원의 기정값과 최대값을 설정한다. 이 자원한계는 실천적으로 유일한 제한이다. 정의된 매 여섯개의 자원한계에 대하여 기정값

을 포함하는 rlim_cur(현재 자원한계)와 체계에 부과된 자원의 최대값을 포함하는 rlim_max(최대자원원천)가 존재한다. Linux 2.4에서 프로쎄스당 열린 파일에 대하여 체계가 설정한 기정한계값은 1024이다. 매 프로쎄스는 어떤 주어 진 순간에 rlim_fd_cur까지만 열려 있을수 있다.

프로쎄스에 대하여 열린파일의 총수는 매개 프로쎄스가 생기자마자 3개의 열린 파일을 가지기때문에 항상 보충적인 3개의 파일을 포함한다. 즉 stdin, stdout, stderr(표준입력, 출력, 오유) 이것들은 모두 프로쎄스를 위한 입력, 출력, 오유출력파일을 의미한다.

shell에 의거하여 ulimit(1) 혹은 limit(1)명령을 리용하면 rlim_fd_cur를 볼수 있다.

sh나 bash를 리용하는 사람들은 ulimit(1)명령을 리용하며 C shell(/bin/csh)를 쓰는 사람들은 limit(1)을 리용하면 된다.

```
[root@hatta /root] # ulimit  -a
core file size (blocks)       1000000
data seg size (kbytes)        unlimited
file size (blocks)            Unlimited
max memory size (kbytes)      Unlimited
stack size (kbytes)           8192
cpu time (seconds)            Unlimited
max user processes            2048
pipe size (512 bytes)         8
open files                    1024
virtual memory(kbytes)        2105343
```

rlim_fd_cur는 “open file”로 현시되는데 여기서는 그의 기정값(가장 많은 배포판에서)은 프로쎄스당 1024로 보여 주었다. 루트에 관하여 자원한계검사를 하지 않을수도 있으며 또 열린 프로쎄스한계를 리론적으로 3억개(signed int형자료의 최대값)까지 설정할수 있다. 분명히 열려 있는 매개 파일에 요구되는 프로쎄스당 파일구조체를 위한 가상주소공간밖에서 프로쎄스가 처리되기때문에 이 값에 가까운 그 어떤 값도 얻을수 없다. 또 이 정도로 많은 파일을 열어야 할 환경에 맞다들리는 일도 물론 없다. 그러나 우리는 프로쎄스당 열린 파일의 개수를 수천 혹은 지어 수만개로 설치할 때도 보게 된다.

동일한 파일이 그와 관련된 다중파일조종자료구조체를 가질수 있다고 생각해 보겠다. 만일 같은 체계상에서 실행되는 서로 다른 프로쎄스들이 같은 파일을 연다면 매개 프로쎄스는 그 프로쎄스를 정의한 파일조종자료구조체에 대한 파일서술자를 통하여 유일한 파일표현을 가지게 된다. 따라서 매개 프로쎄스는 파일을 읽거나 쓸 때 파일조종자료구조체의 f_offset수를 변경시키게 된다. 더우기 그 동작이 fork()나 혹은 clone()체계호출을 거쳐 계층화된 파일서술자와 다르게 되며 또 프로쎄스가 파일서술자의 중복을 위하여 dup()체계호출을 내보낼 때도 달라 진다. 이 순차적서술에서 파일구조체와 f_offset마당은 둘다 공유되며 따라서 파일에 대한 상위프로쎄스의 읽기나 쓰기는 fork(2)의 경우에 하위프로쎄스에서 볼수 있

는것처럼 그 파일의 f_offset를 변경시키거나 혹은 dup(2)와 dup2(2)에 파일서술자에 대한 참조값들을 변경시킨다.

범위에 기초한 배정(일반)

verita의 VxFS나 IBM의 JFS와 같은 범위에 기초한 파일배정체계는 디스크블로크를 범위로 배정한다. 범위는 배정된 다중블로크들의 련속된 렬을 하나의 단위로 하여 설정하며 파일이 처음에 생성되였을 때 시작되는 론리적변위/길이/물리적길이를 포함하는 3중구조로 서술된다. 주소화된 구조체는 범위서술자(3중)를 가지고 위치한 B+나무*이며 색인마디안에 뿌리화되여 있고 파일안의 론리적변위에 의하여 열쇠화된다. 파일체계의 메타자료(meta−data)는 파일이 처음으로 생성될 때 기록되는데 블로크에 기초한 배정과는 다르다. 초기배정이 순서화되여 있기때문에 그 다음에 수행하는 읽기, 쓰기, 찾기도 물론 순서적이여야 한다는것이 강조된다.

첫번째 배정과 블로크범위내에 다음의 범위가 배정될 때까지 보충적메타자료쓰기는 요구되지 않는다. 이 방법은 디스크찾기패턴을 최량화하며 또 쓰기블로크들을 클라스터로 그룹화하는것은 파일체계로 하여금 기억장치에 대하여 수많은 소규모 SCSI전송휴지시간을 절약하면서 보다 큰 물리적디스크쓰기능력을 발휘하게 한다. 블로크에 기초한 배정에서 배정된 파일의 매개 론리적블로크를 위하여서는 매개 파일에 대하여 수많은 메타자료가 생긴다는데로부터 블로크주소번호가 요구된다. 범위에 기초한 배정방법에서는 매개 련속된 자료블로크의 범위에 대하여 오직 출발블로크의 번호와 길이만을 요구한다. 몇개의 아주 큰 범위를 가진 그러한 파일은 작은 량의 메타자료만을 요구한다. 그림 3−5에서 배정도식의 두 방법 즉 블로크배정과 범위배정방법사이의 차이를 보여 주었다.

범위에 기초한 파일체계는 순차적배정방법과 보다 큰 쓰기용블로크의 클라스터링으로 하여 순차파일접근에서 훌륭한 성능을 제공한다.

실례로 범위에 기초한 파일이라고 해도 순차적으로 읽기를 요구하는 경우 시작블로크번호와 그것의 길이만 읽으면 된다. 다음부터는 그 범위내에 있는 모든 자료블로크를 련속적으로 읽을수 있다. 아주 작은 메타자료의 읽기경우에 순차적인 자료의 읽기는 휴지시간을 산생시킨다. 그리고 범위에 기초한 파일체계는 파일체계가 우연 I/O에 리용

---

* B+나무는 물리적주소자료라든가 첨수화된 자료를 포함하는 레코드(잎)를 검사, 검색, 삽입, 지울수 있는 특별한 종류의 m−차균형나무이다. B+나무는 잎마디에 의하여서만 지적되는 자료를 가지며 가상적인 열쇠를 리용하여 한번이상 출현하는 검색열쇠도 가질수 있다. 보통 B+나무는 마디의 크기를 페지크기에 의하여 정의하는 방법으로 만든다. 탐색경로가 릉을 따라 가야 할 때는 페지화가 요구된다. 자료는 잎에만 기억되며 마디에는 기억되지 않는다. 다만 일명 "read map"라고 하는 참조열쇠잎만은 마디에 보존된다. B+나무는 열쇠를 효과적으로 찾아 낼수 있는 가장 좋은 구조이다. 그러나 실제적인 문제는 그것이 수행하는 3가지 서로 다른 과제로부터 즉 검색, 삽입, 지우기로부터 발생한다. 지우기와 검색이 병행적으로 진행되는 경우에 작업의 완료와 완전정지에 빠지는 경우는 50%정도이다. 그러므로 병렬화는 범위에 기초한 파일배정체계에서는 그다지 좋은 성능최량화방법이 되지 못한다.
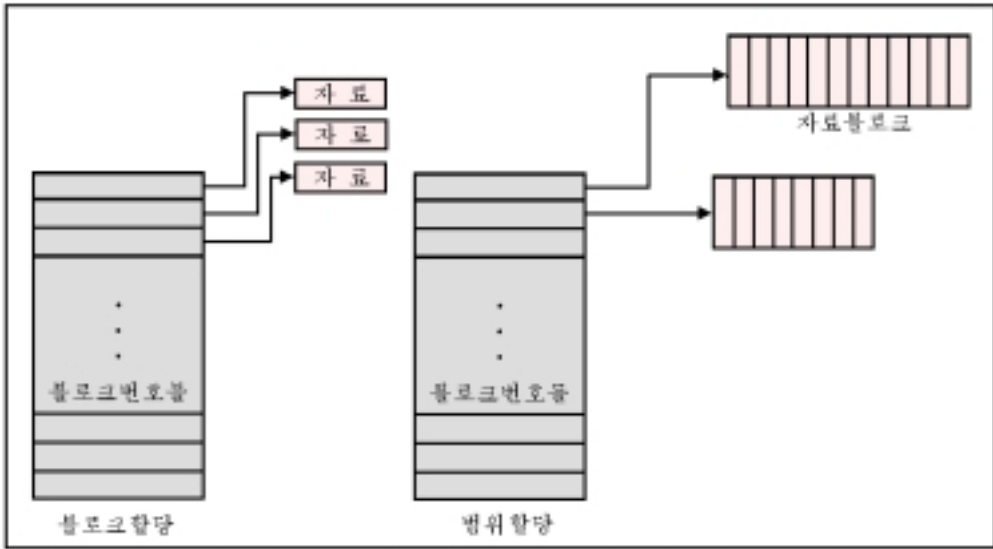
그림 3-5. 블로크배정과 범위배정

될 때에는 효과를 나타내지 못한다. 파일을 우연방식으로 읽을 때는 우리가 블로크에 기초한 파일체계에서 한것과 류사한 방법으로 매개 자료블로크읽기를 위하여 요구된 블로크의 블로크주소를 검색하여야 한다.

몇가지 파일체계와 그것들의 배정방법을 아래의 표에 보여 준다.

표-1　　　　여러가지 파일체계와 그것들의 배정양식

파일체계	할 당 방 법
Ext2	블로크에 기초한 배정, 배정자는 련속블로크들을 배정한다.
ReiserFS	범위에 기초한 배정
JFS	범위에 기초한 배정

블로크에 기초한 배정(일반)

전통적인 UNIX파일체계에서 리용한 배정방식은 블로크기초배정방식이다. 파일이 확장되는 경우에 이 방법은 파일에 대한 최소블로크수(파일체계에 의하여 정의된바와 같이)를 먼저 배정한다. 블로크들은 자유블로크로부터 배정된다. 이 방법으로 기억공간을 보존하려고 할 때 블로크들은 때로 우연순서로 배정된다. 이것은 과도한 디스크찾기를 발생시키며 따라서 파일체계로부터의 읽기에서는 파일의 확장에 따라 배정되는 모든 우연블로크위치를 찾아 내기 위한 디스크기술문제가 제기된다.

우연블로크배정은 블로크배정방법을 련속된(순서적으로) 블로크를 배정하는 방법으로 최량화하여 극복할수 있고 대략적인 순차배정은 보다 재치 있는 블로크배정수법을 리

용하여 진행할수 있으며 이때 디스크의 찾기시간은 훨씬 감소된다. 하지만 련속된 파일체계블로크배정은 파일체계전반에 걸쳐 파일블로크들을 토막화하는 결과를 초래하게 될것이며 파일체계접근은 결국 우연속성으로 되돌아 오게 될것이다. 블로크배정도식은 매번 블로크가 확장될 때마다 새로운 블로크배정에 대한 정보를 기록해야 한다. 어떤 시각에 한개 블로크가 확장될 때에는 많은 여유디스크 I/O가 파일체계블로크조종구조체를 기록해야 한다(파일체계블로크조종구조체의 정보는 메타자료를 말한다.).

ext2파일체계에서 메타자료는 기억장치에 비동기적으로 씌여 지며 따라서 파일의 크기를 변경시키는 조작들은 매 메타자료조작의 완성을 기다릴 필요가 없다. 이것은 파일의 갱신속도를 크게 개선시키지만 실제적으로 메타자료를 쓰기전에 파일내용이 변경되여 체계폭주가 발생하며 모순된 자료로 인한 위험성이 커지게 된다.

거래처리 혹은 자료기지안전문제

다중프로쎄스나 혹은 다중 스레드로부터 같은 파일에 대한 동기화된 접근을 관리할 목적으로 파일잠금대면부가 제공된다. 이 대면부들은 최근의 서적 Linux File System에서 론의된다. 련관된 관점으로 볼 때 Linux 혹은 임의의 UNIX형 OS는 실제적으로 파일레코드표식에 대한 핵심부준위의 지원은 제공하지 않는다. 이 절에서는 정적형, 실행기록(Journaling)(실행기록형) 그리고 사용등록형(logging)파일체계들사이의 차이점에 대하여 고찰한다. 모든 자료기지파일들은 파일체계에 상주되여야 한다. Linux의 고유파일체계 즉 ext2는 3~4년전까지는 여러가지 작업을 원만히 해냈지만 현재 Linux시장의 도전에 원만히 대응해 나가지 못하고 있다. 그 리유의 하나가 바로 ext2가 정적이라는데 있다. 즉 Linux는 디스크에 대한 모든 갱신작업을 일정하게 진행했다는것을 담보할수 있게 변경사항을 기억하지 못한다.

더우기 ext2fs는 메타자료를 비동기적으로 기록하는 임의의 OS우의 몇가지 파일체계중의 하나이다. 이것은 파일연산을 상당한 정도로 가속화하는 한편 생성, 변경날자, 소유권, 허가 등과 같은 파일에 대한 정보가 파일내용과 관련하여 지연된 형식으로 기록된다는것을 말한다. 파일에 대한 갱신을 기록한 다음 실제적으로 파일머리부를 쓰기전에 성능이 저하되면 여기에는 문제가 있는것으로 간주한다.

ext2fs의 이러한 부족점은 Linux를 자료기지봉사기로 광범히 리용하는데서 주되는 장애로 되고 있다. 실례로 Linux용 Oracle8i는 원시I/O를 지원하지 못한다(2.2x핵심부로부터는 지원된다.).

Linux해커들중에는 ext2fs의 이러한 부족점을 고려하여 실행기록이나 사용등록파일체계를 리용하여 해석하려는 시도도 나왔다.

비실행기록파일체계에 비한 실행기록파일체계의 우점

여기서 고찰하는 실행기록파일체계 즉 JFS는 체계폭주될 때 파일체계의 재기동시간을 단축하기때문에 인터네트파일봉사기의 기본기술이다.

자료기지상태기록기술을 리용할 때 JFS는 몇s, 몇min내로 체계파일을 정상상태로 회복할수 있다. 비실행기록형파일체계에서는 파일을 복구하는데 몇시간 혹은 며칠씩 걸린다. 그러므로 실행기록형으로 이전시키는 기술에 의하여서만 체계파일들이 파일을 정상상태로 검

증/회복할수 있도록 파일체계의 모든 메타자료검사에 필요되는 시간을 줄일수 있다.

ext2와 같은 정적파일체계에는 색인마디위치에 관한 배치도가 있다. 이 색인마디들은 다른 색인마디나 매개 파일이름과 관련된 색인마디들의 표를 포함하는 등록부블로크들과 자료블로크들을 지적한다.

임의의 UNIX등록부와 마찬가지로 Linux등록부는 파일이름과 색인마디번호사이의 관계이다. 파일의 색인마디번호는 ls명령에 "−i"스위치를 주어 찾아 볼수 있다. 색인마디는 디스크상에 파일에 관한 자료블로크가 어디에 있는가를 지적하는 지적자와 함께 소유권과 허가정보를 포함한다. 이제 어떤 파일 "test.file"의 내용을 변경시킬 때 어떤 일이 생기는가를 보자. "test.file"을 위한 색인마디가 4개의 자료블로크를 목록화하고 있다고 가정하자.

"test.file"의 자료는 디스크위치 3110, 3111, 3506, 3507에 존재하고 있다. 디스크블로크의 초기 배정시에 3111과 3506사이에는 다른것들이 이미 배정되였으므로 간격이 생기게 된다. 이때 우리는 이 파일이 토막화된다고 본다. 하드구동기는 디스크면우에서 3110구역을 찾아야 하며 거기서 두개 블로크를 읽고 다음 3506구역을 찾고 전체 파일을 읽기 위하여 두개의 블로크를 읽어야 한다.

이제 세번째 블로크를 변경시켜 보자. 변경에 따라 파일체계는 세번째 블로크를 읽고 재쓰기를 진행하는데 여전히 3506을 가리킨다. 파일을 첨가하면 임의의 곳에 블로크가 배정될수 있다. 전원이 불결하면 위험이 동반된다. 등록부의 갱신동작이 절반단계에서 진행되고 있다고 가정하자. 큰 등록부의 다섯번째 블로크에 있는 23개 파일입구점을 교정한다. 디스크가 이 블로크에 대한 쓰기작업도중에 있기때문에 전원이 중단되면 블로크는 완성되지 못하며 따라서 자료가 손상되게 된다.

재기동시 Linux(모든 UNIX와 같은)는 fsck(파일체계검사)프로그람을 실행한다. 이 프로그람은 전체 파일체계에 걸쳐 모든 입구점들의 정확성을 검증하고 블로크들이 정확하게 배정되고 참조되는가를 확인하는 작업을 단계적으로 수행한다.

이 프로그람은 손상된 등록부의 입구점들을 찾고 그것을 재생하려고 하지만 fsck가 실제적으로 손상회복을 관리한다는 확신성은 어디에도 없으며 흔히 실제적으로 그렇게 되지 않는다. 때때로 우에서 설명한것과 같은 조건에서 모든 등록부입구점들은 루실될수 있다. 큰 파일체계들에서 fsck의 실행에는 아주 긴 시간이 걸릴수 있다. 수기가바이트정도의 파일을 가진 기계에서 fsck는 10h 혹은 그이상 시간까지도 수행될수 있다. 물론 이 시간동안 체계를 리용할수 없으며 이것은 허용할수 없는 휴식시간으로 된다.

실행기록형파일체계는 이 문제를 해결하지만 또 새로운 문제를 산생시킨다. 어떻게 발생하며 왜 그렇게 되는가를 보자.

실행기록형파일체계의 동작

실행기록형파일체계(JFS)는 초기에 자료기지가 파일체계메타자료에 대하여 수행되는 조작에 대한 정보를 최소거래로 등록하도록 개발된 기술이다. 체계고장사건시 파일체계는 가동일지(log)들의 재생과 적당한 거래에 가동일지기록을 적용하는 방법으로 정상상태로 회복된다. 이러한 가동일지기초형방법과 련관된 회복시간은 재생편의프로그람이 파일체계메타자료 전체를 검사하지 않고 최근의 파일체계동작에 의하여 생성된 가동일지기록

만을 검사해야 하기때문에 대단히 빠르다.

JFS와 같은 실행기록형파일체계는 구조적정확성과 회복성이 개선되였으며 HPFS, ext2 그리고 전통적인 UNIX파일체계와 같은 비실행기록형파일체계보다 회복시간이 매우 빠르다(JFS는 수s 혹은 수min내에 체계파일을 정상상태로 회복한다.).

다른 체계파일들은 론리적파일쓰기조작이 흔히 다중매체I/O에서 이루어 지게 되고 전체적으로 어떤 주어 진 시간내에 반영되지 않기때문에 체계고장사건시에 디스크자료손상이 있게 된다.

이러한 파일체계들은 등록부와 디스크주소구조체와 같이 구조적인 종합문제들을 검출하고 회복하기 위한 모든 파일체계의 메타자료를 검사하는 재시동-시간편의프로그람들(Linux에서는 보통 fsck를 의미한다.)에 의존한다. 이 편의프로그람은 극단한 경우에 자료를 잃거나 틀리게 할수 있는 시간소비지향 및 오유지향성프로쎄스로 될수 있다. 실행기록형파일체계들은 색인마디의 변경만을 계속 보존하지만 파일의 내용은 변경시키지 않는다. 따라서 파일체계의 사용등록은 자료와 색인마디에서 이루어 진 변경들을 둘다 계속 보존한다 . 모든 변경들과 추가, 지우기들은 가동일지처리로 알려 진 파일체계의 특정한 부분에 등록된다.

“test.file”에 의한 첫번째 실례에서는 3506블로크에 있는 자료를 변경시키지 않고 등록파일체계(log file system)가 “test.file”과 디스크상의 새 위치에 있는 세번째 블로크의 색인마디들의 복사를 기억하게 된다.

색인마디들의 기억목록은 “test.file”을 새로운 색인마디로 지적하게끔 변화된다. 이따금 파일체계는 디스크상의 색인마디목록을 검사하거나 갱신하며 리용되지 않은 파일부분을 공간으로 남긴다(“test.file”의 첫 세번째 블로크).

등록파일체계는 디스크의 동일한 구역에 대하여서만 추가동작을 하며 그우에서 블로크들을 여기저기 찾을 필요가 실제적으로 없기때문에 쓰기속도가 비약적으로 높아 진다. 그러므로 쓰기속도가 개선되며 회복시간도 역시 줄어 든다(실제적으로 구조적특성으로 하여 회복시간이 전혀 없다.).

정적파일체계에 비하여 등록파일체계는 파일자료블로크들을 쉽게 찾을수 있기때문에 거의 동일한 읽기속도를 가진다. 색인마디배치도는 블로크들의 목록으로 구성되고 목록들은 보통 기억넘기기되기때문에 빨리 읽을수 있다.

따라서 등록파일체계는 가장 훌륭한 측면을 가지고 있다고 볼수 있다. 읽기시간은 정적파일체계와 거의 동등하다(토막이 많기때문에 아마 비트속도는 더 늦어 질수 있다. 등록파일체계에서 중요한 문제는 쉽게 토막화될수 있다는데 있다.).

등록파일체계의 일반적형태

파일체계가 디스크구조체로 변화될 때는 변화시키는데 필요한 몇개의 차단된 동기적인 쓰기를 리용한다. 만일 조작도중에 중지가 발생하면 파일체계의 상태는 미정으로 되며 전체 파일체계에 대하여 일관성을 검사해야 한다. 만일 한개 블로크를 파일의 끝에 추가할 때 파일의 매 블로크가 어디에 위치하고 있는가를 통지하는 디스크넘기기가 자료블로크를 쓰기전에 디스크를 읽거나 변경시켜야 하며 재기록해야 한다.

오유가 발생할 때 파일체계는 기동하여 올려태우기되기전에 검사되여야 하는데 이때 파일체계관리기는 블로크넘기기가 정확한지 알지 못하며 또 폭주시에 어느 파일이 변경

되였는지 모른다.

이때는 완전한 파일체계주사동작으로 넘어 간다. 메타자료등록파일체계는 디스크상에 wrap-around추가전용등록구역을 가지고 있다.

이 구역에서 파일체계는 매개 디스크거래의 상태를 등록하는데 리용한다. 어떤 디스크구조체가 변경되기전에 intent_to_commit레코드가 가동일지에 기록된다. 이때 등록부구조체는 갱신되며 가동일지입구점은 완전히 표식화된다. 파일체계구조체에 대한 매개 변경이 가동일지에 있기때문에 완전파일체계주사를 요구하지 않고도 가동일지를 보고 파일체계의 일관성을 검사할수 있다. 올려태우기시에 indent_to_commit입구점이 보이지만 완전히 표식화되지 않았으면 블로크용파일구조체가 검사되며 필요한 경우 고쳐 진다.

그림 3-6은 정규파일(정적)체계 즉 ext2fs에서 파일의 자료블로크와 색인마디정보(변경시간, 자료블로크에 대한 지적자 등) 그리고 파일에서 자료를 변경시킬 때 어떤 일이 발생하는가를 보여 준다.
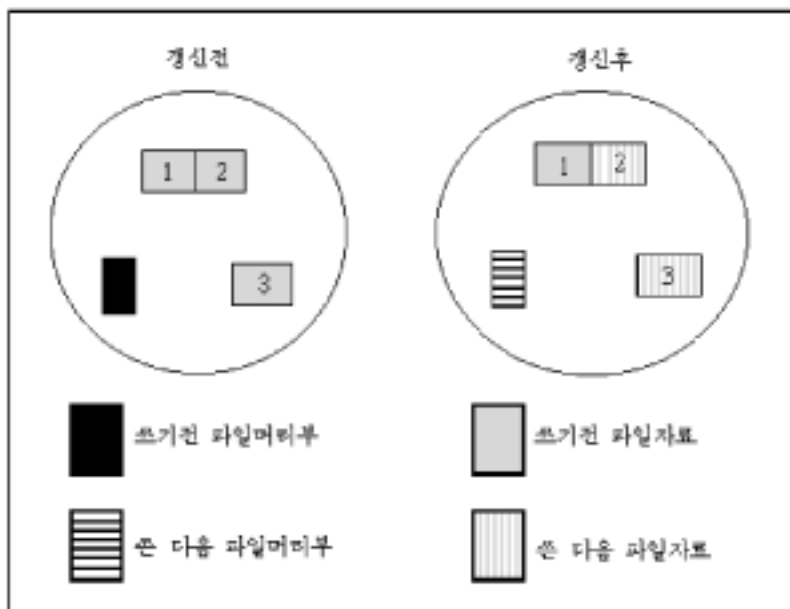
그림 3-6. Ext2파일체계에서 갱신조작

그림 3-7은 등록파일체계에서의 류사한 파일과 그것이 변경될 때 어떤 일이 생기는가를 보여 준다.

변화되는 매개가 가동일지의 끝에 추가된다는것을 강조한다. 이 방법은 파일부분들에 쓸 때 디스크전체를 찾을 필요가 없기때문에 속도가 빠르다.

또한 가동일지가 《새》자료블로크들을 성공적으로 쓸 때까지 파일의 초기자료블로크를 《잊어 버리지》않기때문에 아주 안전하다. 이것은 폭주후에 fsck시간이 거의 필요없이 아주 안전한 파일체계를 유지할수 있게 한다. 이러한 파일체계들은 마지막검사점이 검사된 후 파일체계만을 갱신하기때문에 폭주후 거의 즉시적으로 직결상태로 돌아 온다.

가동일지안에서 모든 변경이 빨리 재적용되여 디스크의 손상된 부분은 언제나 가동일지에
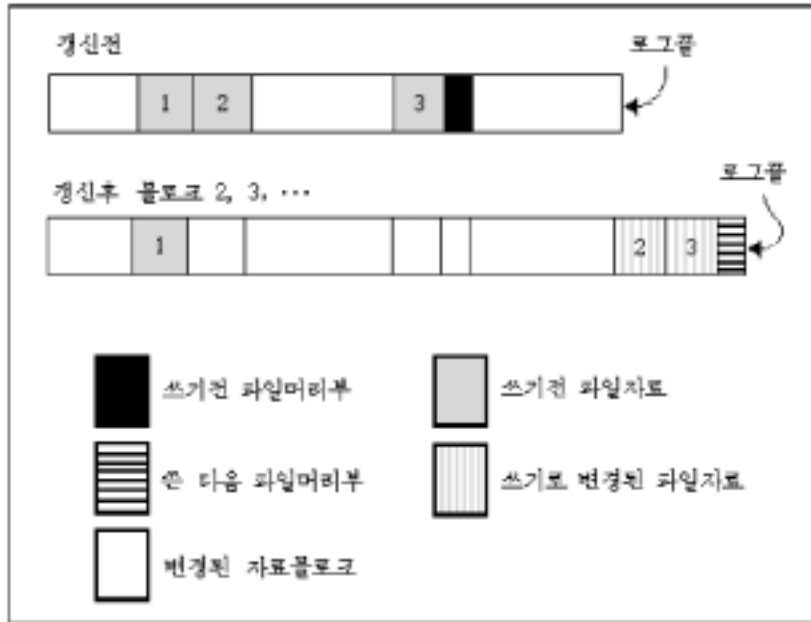
그림 3-7. 사용등록파일체계에서 갱신조작

첨가된 마지막변경으로 된다. 이 변경내용은 무효한것이므로 버릴수 있다. 전원차단시에 잃어 진 자료는 하나의 내용만 변경된다.

20GB의 메타자료등록파일체계와 그이상의 체계들에서는 대표적으로 폭주후 fsck와 회복에 4~5s정도 걸린다. 따라서 메타자료등록파일체계는 몇초동안에 올려태우기되는 아주 대중화된 파일체계와 수십 혹은 수시간 올려태우기되는 등록 없는 체계사이에 차이가 생기게 한다. 그럼에도 불구하고 사용등록은 자유롭지 못하며 여기에도 무시할수 없는 성능상의 휴지시간이 존재한다는데 대하여 주의를 돌려야 한다.

사용등록은 느린 동기적쓰기들을 더 많이 요구하며 가장 일반화된 등록의 실현인 메타자료등록도 파일갱신에 적어도 3개의 쓰기를 요구한다. 이것은 사용등록없이 진행할 때보다도 더 많다는것을 의미한다.

결과적으로 우리가 요구하는것이 무엇인가에 대하여 주의를 돌려야 한다. 즉 파일체계의 속도를 빨리 할것을 요구하는가 혹은 실현성을 최대로 할것을 요구하는가에 대하여 주의를 돌려야 한다.

몇가지 등록파일체계들은 가동일지에 메타자료와 함께 파일자료로 설정하는 선택항목을 제공하고 있으므로 여기서는 두번찾기와 쓰기를 피할수 있다.

자료는 처음에 가동일지에 씌여 지고 다음에 파일체계안에 반복한다.

이 조작은 두가지 일을 수행한다.

우선 포함되지는 않지만 마지막블로크까지 씌여 진 자료의 완전성을 확인한다.

또한 전자우편이나 새 소식봉사기에서와 같이 소규모동기쓰기의 성능을 발휘하게 한다. 그렇지 않은 경우에는 매개의 응용프로그람쓰기에 대하여 디스크의 서로 다른 부분에 둘 혹은 그이상의 쓰기가 요구된다(자료를 위한 쓰기와 사용등록기록을 위한 쓰기).

제4장. 가상파일체계 VFS

이 장에서는 Linux가상파일체계(VFS)에 대하여 3장에서 일반적으로 고찰한 내용들을 보다 상세하게 취급한다.

Linux파일체계에 대하여 정확히 리해하자면 핵심부와 대면부에 대하여 완전히 파악하여야 하는데 그러한 대면부가 바로 가상파일체계이다.

UNIX형OS에서와 같이 Linux는 VFS를 가지는데 오랜 시간이 걸렸으며 Free BSD조작체계들을 비롯하여 Solaris, AIX, HP_UX들도 모두 류사한 VFS를 가지고 있었다. 여기에서 주목되는것은 Linux만이 가장 최신형의 VFS를 가지고 있다는것이다.

Linux계통의 연구자들은 오래동안 프로그람을 작성하고 많은 의견들을 종합처리하여 설계를 완성함으로써 새로운 판본 2.4.0으로서 LinuxVFS를 완성하였다.

일 반 적 개 념

Linux에서 모든 파일들은 가상파일체계스위치 혹은 VFS에 의하여 접근된다. VFS는 일반적파일체계기능과 요청을 조종하기 위하여 정의된 정확한 코드들에 대한 벡토르요청을 실현하는 코드의 한 계층이다. 임의의 형태의 파일체계들은 블로크장치접근을 실현하기 위하여 VFS를 리용한다.

VFS의 원천코드

제2장 《Linux 핵심부콤파일》에서 고찰한바와 같이 VFS의 원천코드는 캐쉬와 실행가능한 모든 파일형식을 취급하는 코드와 같은 몇가지 관련된 부분들과 함께 Linux핵심부의 원천부분등록부 fs/에 존재한다. 매 정의된 파일체계는 보다 낮은 부분등록부에 보관되는데 례를 들어 ext2파일체계원천코드는 fs/ext2/에 보존된다.

다음의 표는 fs부분등록부의 파일이름과 그 매개의 기본목적을 설명한다. 중간렬 즉 표식화된 체계는 파일이 결정되는(기본적으로) 기본부분체계가 어느것인가를 보여 주었다. EXE는 실행가능한 파일을 인식하고 적재하는데 리용된다는것을 의미한다.

DEV는 장치구동기지원에 리용된다는것을 의미하며 BUF는 캐쉬를 의미한다.

VFS는 VFS의 한 부분이며 파일체계정의코드에 관한 몇가지 기능성을 대표한다는것을 의미한다.

VFS는 또한 이 코드가 완전히 고유하며 정의된 파일체계코드에 관한 그것의 조작부분을 절대로 대신할수 없다는것을 의미한다.

하지만 파일체계를 쓰는데서 이러저러한 걱정은 하지 않아도 된다.

파일이름	체계	목　　적
binfmt_aout.c	EXE	실행가능한 변경된 형식의 a.out를 인식하고 실행
binfmt_elf.c	EXE	실행가능한 새로운 ELF를 인식하고 실행
binfmt_java.c	EXE	Java apps와 applets를 인식하고 실행
binfmt_script.c	EXE	#!-형식스크립트를 인식하고 실행
block_dev.c	DEV	블로크장치를 위한 함수 fsyn(1), 일반 read(), write()
buffer.c	BUF	블로크화된 캐쉬를 블로크장치로부터 읽는 캐쉬
dcache.c	VFS	등록부이름을 검색하는 등록부캐쉬
devices.c	DEV	등록기와 같은 고유장치지원함수
dquot.c	VFS	고유디스크배정코드지원
exec.c	VFS	고유실행가능지원 .binfmt_*files의 함수호출
fcntl.c	VFS	fcntl()조종
fifo.c	VFS	fifo조종
file_table.c	VFS	체계상의 열린파일의 동적확장가능목록
file systems.c	VFS	모든 콤파일된 파일체계가 init_name_fs()호출에 의하여 초기화된다.
inode.c	VFS	체계상의 열린 색인마디의 동적확장가능목록
ioctl.c	VFS	ioctl을 위한 첫 단계조종. 조종은 파일체계 혹은 필요하면 장치구동기에 넘겨 진다
locks.c	VFS	fcntl()잠금, flock()잠금, 위임잠금
namei.c	VFS	경로이름에 주어 진 색인마디에 채우기, 몇개의 이름관련체계호출을 수행
noquot.c	VFS	배정없음. dquot.c의 #ifdef를 피하기 위한 최량화
open.c	VFS	open() , close(), vhangup()을 포함하는 여러개의 체계호출
pipe.c	VFS	파이프들
read_write.c	VFS	Read(), write(), readv(), write(), lseek()
readdir.c	VFS	등록부를 읽기 위한 여러가지 서로 다른 대면부
select.c	VFS	select()체계호출의 quts
stat.c	VFS	Stat()와 readlink()지원
super.c	VFS	상위블로크지원 파일체계레지스트리, mount()/umount()

VFS의 동작

앞에서 언급한바와 같이 Linux가상파일체계는 아무것도 하지 않지만 파일체계와 관련되는 모든 체계호출을 중간에서 취하는 하나의 계층이다. 그러므로 VFS는 모든 파일체계에 대한 표준적인 대면부(혹은 API)를 제공해야 한다. 사실 Linux상에서 ISO9660CD-ROM이나 DOS형식플로피디스크의 내용을 목록으로 연시할 때(즉 ls)나 /boot등록부를 보기 위한 명령들사이에는 아무런 차이도 없다.

이러한 공통적인 대면부를 얻기 위하여 VFS는 공통파일모형을 도입하였다. 이 모형은 응용프로그람에 대하여서는 모든 파일체계들이 서로 꼭 같은것으로 볼수 있게 한다. 이 모형은 또한 ext2, JFS 혹은 다른 그 어떤 파일체계든지 개별적파일체계들에 이르기까지 VFS의 요구에 대한 파일의 리해를 서로 번역할수 있게 해준다.

VFS공통파일모형에서 등록부들은 마치 다른 파일이나 다른 등록부의 목록을 포함하고 있는 파일인것처럼 보인다. 몇가지 파일체계 즉 FAT, FAT32, VFAT와 같은것들은 등록부나무안의 매 파일의 위치를 기억하며 그 경우에 등록부들은 표준파일로 되지 않는다. 따라서 이 파일체계들은 파일체계상에서 동작할 때의 파일처럼 동적으로 등록부의 보기기능을 실현해야 한다.

분명히 그러한 보기기능들은 파일체계안에는 기억되지 않으며 핵심부의 공간에 객체로서만 존재한다.

더 론의하면 Linux는 실제적으로 실제적핵심부함수안으로 파일관련체계호출을 실행하지 않는다. 한편 read()나 ioctl()은 제기된 문제에서 파일을 조종하는 매개 파일체계에 대하여 정의하는 함수에 대한 지적자로 변환된다.

이 기능을 실현하기 위하여 VFS공통파일모형은 매개 파일체계가 UNIX의 전통적방법에 대응시키는 일련의 파일체계조종블로크들에 대한 표기를 도입하였다.

이 조종블로크들과 구조체들에는 다음과 같은것들이 있다.

상위블로크구조체	올려태우기된 파일체계에 대한 정보. 이 구조체는 보통 장치에 기억된 파일체계조종블로크를 정합시킨다.
객 체	정의된 파일에 대한 정보. 보통 이 정보는 디스크상의 파일조종블로크를 지적한다. 이 매개 객체들은 색인마디번호를 가지며 이 번호는 해당 파일체계안의 파일을 유일적으로 지적한다.
파일객체	현재 열려 진 파일과 거기에 접근하는 프로쎄스에 대한 정보. 이 구조체는 핵심부에만 보존되여 있으며 파일닫기상태에서는 나타나지 않는다.
덴트리객체	등록부의 입구점과 관련된 파일내의 련결에 대한 정보. 매개 파일체계에서 이 련결실현과정은 서로 다르다.

앞에서 본바와 같이 VFS는 공통계층을 생성하는데만 제한하지 않는다. VFS는 덴트리캐쉬를 리용하여 파일체계의 성능을 최대로 높일수 있게 한다. 이 캐쉬는 파일자체의

색인마디에 대한 경로이름을 아주 빨리 검색할수 있는 방법을 제공한다. 통계적으로는 최근에 접근된 파일들이 보다 먼저 다시 요구될 확률이 더 클것이며 덴트리캐쉬는 이 처리를 매우 가속화한다.

VFS는 또한 디스크캐쉬도 유지관리한다. 이 캐쉬는 가상기억에 기억되는데 이 가상기억에는 디스크캐쉬에 기억된 정보의 리용을 고속화할수 있도록 몇개의 구조체가 디스크상에 표준적으로 기억된다. 덴트리캐쉬외에 Linux에는 캐쉬와 페지에서의 캐쉬도 있다. 두개의 캐쉬는 앞부분에서 설명하였다.

VFS-조종체계호출

파일이나 파일체계를 취급하는 일부 체계호출은 추상화된 형식을 함수호출을 조종하는 실제적파일체계정의로 변환할수 있도록 VFS가 중간에서 선취할것을 요구한다.

다음의 표에서 지적한것외에 bind(), connect(), ioperm(), ioctl(), pipe(), socket(), mknod()체계호출들도 역시 VFS층에 의하여 조종된다.

체 계 호 출	기 능
chroot()	뿌리등록부를 변경
chmod(), fchdir(), getcwd()	현행등록부를 변경
dup(), dup2(), fcntl()	파일서술자조종
fdatasync(), fsync() ,sync(), msync()	완충기의 동기화
lseek(), _lseek()	파일변위를 변경
mount(), umount()	파일체계의 올려태우기와 올려태우기해제
getdents(), readdir(), link(), unlink(), rename()	련결조종
readlink, symlink()	쏘프트련결조종
read(), write(), readv(), writev(), sendfile()	파일 I/O
pread(), pwrite()	파일찾기와 I/O조종
mmap(), munmap()	기억에로의 파일넘기기
flock()	파일잠금
stat(), fstat(), lstat(), access()	현재파일상태에 접근
truncate(), ftruncate()	파일의 크기변경
open(), close(), creat(), umask()	파일의 열기와 닫기
select(), poll()	비동기 I/O조종
sysfs()	파일체계에 대한 일반적정보
statfs(), fstatfs(), ustat()	파일체계에 대한 통계자료읽기

이것들에 대해서는 이 장의 뒤부분에서 따로 설명한다.

덴트리캐쉬

앞부분에서는 핵심부안의 가상기억관리기가 블로크장치용 두개의 캐쉬 즉 완충고속기억기와 폐지에서의 고속기억기를 어떻게 관리하는가에 대하여 보았다. Linux의 VFS층에 의하여 관리되는 캐쉬들중의 하나가 덴트리캐쉬이다. 파일이름에 대한 대응하는 색인마디의 검색은 장치에로의 접근을 요구하며 특히 경로이름이 복합화된 경우에는 성능상의 견지에서 매우 경제적이다. 이 사실은 가장 최근에 검색된 파일의 캐쉬와 경로 그리고 색인마디번호들을 관리해야 할 요구를 제기하고 있다. 이 캐쉬를 바로 덴트리캐쉬라고 부른다.

덴트리캐쉬는 두개의 주요자료구조체로 구성된다.

▼ 사용되였거나 사용되지 않은 부의 상태에 있는 덴트리객체들의 목록

▲ 파일이름에 대한 덴트리객체들과 그것의 등록부를 빨리 찾기 위한 하쉬표

Linux에서 덴트리캐쉬함수는 역시 색인마디캐쉬이다. 덴트리캐쉬의 색인마디는 여전히 사용되지 않은 덴트리와 관련되며 지워 지지 않고 쉽게 보존된다.

《USED》라는 기발로 표시되는 모든 덴트리들은 련관된 색인마디객체의 덴트리마당에 대하여 지적되는 2중련결목록으로 유지된다. 이 덴트리들은 한점 혹은 다른 점에서 《부》로 될수 있는데 이것은 마지막고정련결이 제거될 때이다. 이러한 상태가 발생할 때 객체는 LRU목록으로 옮겨 진다. LRU목록은 모든 개별화된 덴트리객체에 대한 지적자를 포함하는 시간에 따라 정렬된 2중련결목록이다.

핵심부의 VM관리기가 덴트리캐쉬를 줄여야 한다는것을 결정하면 덴트리캐쉬는 LRU목록으로부터 가장 오래된 덴트리객체를 지우는것으로부터 시작한다.

덴트리객체를 취급하는 원천코드안의 함수들은 덴트리함수라고 부른다. 이것들은 모두 dentry_operation자료구조체를 리용한다. 그것들의 주소는 d_op마당으로부터 얻어 낼수 있다.

몇가지 파일체계 (NFS와 같은)는 자기의 고유한 덴트리조작을 수행한다. 이 경우에 우에서 언급된 파일들은 비워 지게 되며 VFS는 그대신 기정기능을 수행한다.

덴트리캐쉬가 리용하는 기본함수들은 다음과 같다.

덴트리캐쉬기능	목 적
d_compare(dir, name1,name2)	인수에 주어 진 두개의 파일이름을 비교한다. 이 함수는 파일체계의존형이다. FAT파일체계는 비교시 대문자와 소문자를 구별하지 않는다.
d_delete(dentry)	덴트리에 대한 마지막참조를 제거한후 이 함수가 호출된다.
d_hash(dentry,hash)	특정한 덴트리객체에 대한 하쉬값을 계산한다. 이 함수는 파일체계의존형이다.
d_iput(dentry, ino)	덴트리를 《부》로 표시하고 LRU에로 그것을 옮긴다.
d_release(dentry)	덴트리객체를 목록에서 지운다.
d_revalidate(dentry)	변환에 리용하기전에 특정한 덴트리가 아직 유효한 상태에 있는가를 검사한다.

파일체계올려태우기

핵심부는 기동시나 혹은 올려태우기명령에 의하여 표준조작을 진행하는 과정에 새로운 파일체계를 등록한다. 이미 본바와 같이 올려태우기는 VFS층에 의하여 실행되고 있는 이러한 명령들중의 하나이다. 매개 파일체계는 그자체의 뿌리등록부를 가져야 한다(이것은 고리귀환장치들에서도 마찬가지이다.).

init_name_fs()를 위한 임의의 파일체계의 코드를 보면 대체로 한행정도의 코드를 포함하고 있다는것을 알수 있다. 실례로 extfs에서 다음과 같이 된다(fs/ext2/super.c로부터).

```
int init_ext2_fs(void)
{
  return register_file system(&ext2_fs_type);
  }
```

이 조작은 분명히 fs/super.c에 보존된 레지스트리에 의하여 체계파일을 등록하는 조작이다.

ext2_fs_type는 실지로 아주 단순한 구조체이다.

```
static struct file_system_type ext2_fs_type={ext2_read_
         super, “ext2”,1, NULL
  };
```

ext2_read_super입구점은 파일체계가 올려태우기된후의 함수에 대한 지적자이다. 앞의

실례에서 “ext2”은 파일체계의 형이름인데 이것은 장치를 올려태우기할 때 어느 파일체계를 리용하겠는가를 결정하는데 리용된다. 보통 체계관리기는 “_t”선택을 가진 올려태우기명령으로 파일체계를 지적한다.

실례로 mount_t jfs/dev/sda1/disk1은 JFS파일체계를 지적한다. VFS는 올려태우기될 상위블로크로부터 체계파일을 검출할수 있는 능력을 가지고 있다. 따라서 mount_a/dev/cdrom/mnt은 대다수의 핵심부들에서 리용될수 있다. “1”은 장치가 올려태우기될것을 요구하며 (proc.파일체계나 망파일체계에서는 다르다.) NULL은 공간을 채울것을 요구한다는것을 의미한다. 이때 채워 질 공간은 fs/super.c에 보존된 파일체계레지스트리안의 파일체계형의 련결목록을 보존하는데 리용된다. 한개 파일체계가 하나이상의 파일체계형을 지원할수 있다. 실례로 fs/sysv/inode.c에 있는 3개의 가능한 파일체계형은 다음의 코드로 하나의 파일체계에 의하여 지원된다.

```
    static struct file_system_type sysv_fs_type[3]={
         {sysv_read_super, “xenix”,1, NULL},
         {sysv_read_super, “sysv”,1, NULL},
         {sysv_read_super, “coherent”,1, NULL}
};

         int init_sysv_fs(void)
{
         int i;
         int ouch;
         for (i=0; i<3; i++){
         if ((ouch=register_file system(&sysv_fs_type[i]))
          !=0)return ouch ;
         }
         return ouch
}
```

파일체계와 디스크의 련결

핵심부와 파일체계는 오직 파일체계의 형이 부여된 장치가 올려태우기될 때만 서로 작업하기 시작한다. 체계관리기가 ext2파일체계를 포함하는 장치를 올려태우기할 때 super.c(이 장의 마지막에 서술된)로부터 ext2_read_super()가 호출된다. 상위블로크의 읽기가 성공적으로 진행되고 파일체계를 올려태우기할수 있으면 super_operations라고 부르는 구조체에 대한 지적자를 포함하는 정보를 super_block구조체에 써넣는다.

이때 super_operations는 상위블로크와 관련된 공통적조작을 수행하는 함수들에 대한 지적자를 포함하고 있다. 즉 이 경우에는 ext2에 정의한 함수들에 대한 지적자를 말한다.

상위블로크는 장치에 관한 모든 파일체계를 정의하는 블로크이다. 물론 상위블로크를 가지고 있지 않는 파일체계도 있다(FAT체계와 같은).

총체적으로 파일체계에 속하는 조작들은 상위블로크조작으로 고찰할수 있다.

super_operations구조체는 색인마디와 상위블로크를 관리하는 함수들에 대한 지적자를 포함하는데 그것은 전체로서 파일체계의 상태를 참조하거나 변경시킬수 있다(statfs()와 remount()).

유감스럽게도 핵심부개발은 많은 지적자들을 동반하기때문에 파일체계개발자들은 가상파일체계의 능력을 그 지적자들을 정확히 해결하는데 힘을 넣어야 한다.

새로운 파일체계에 대하여 프로그람작성자가 요구하는 모든 내용은 함수에 대한 지적자로서 구조체(보통정적)에 들어 가 있으며 이 구조체에 대한 지적자는 VFS로 다시 넘겨 지며 따라서 파일체계와 파일에서 그것을 얻어 낼수 있다.

실례로 상위블로크구조체는 VFS에서 다음과 같이 볼수 있다.

```
(<linux/fs.h>로부터)
struct super_operations {
     void(*read_inode) (struct inode*) ;
     int(*notify_change) (struct inode*, struct iattr*) ;
     void(*write_inode) (struct inode*) ;
     void(*put_inode) (struct inode*) ;
     void(*put_super) (struct super_block*) ;
     void(*write_super) (struct super_block*) ;
     void(*statfs) (struct super_block*, struct statfs*, int) ;
     int(*remount_fs) (struct super_block*, int*, char*);
};
```

fs/ext2/super.c에 서술된 ext2에서의 선포문에 대하여 비교해 보자.

```
static struct super_operations ext2_sops= {
     ext2_read_inode,
     NULL,
     ext2_write_inode,
     ext2_put_inode,
     ext2_put_super,
     ext2_write_super,
     ext2_statfs,
     ext2_remount} ;
};
```

일부 사용자들의 경우 NULL입구점이 무엇인가에 대하여 대체로 의문을 가질것이다. Linux전반에 걸쳐 함수지적자를 얻기 위하여 어떤 기정동작이 요구될 때마다 NULL지적자는 그 요구를 만족시킬수 있는 편리한 수단이다.

우의 선포문에서 문장구분이 얼마나 간단하고 명백한가를 알수 있다.

sb−>s_op−write_super(sb)와 같은 모든 코드들은 VFS의 실현부안에 숨겨 져 있다.

파일체계들이 상위블로크를 포함하여 디스크로부터 실제적으로 블로크들을 어떻게 읽고 쓰는가에 대한 구체적인 내용은 이 책의 마지막장들에서 론의한다. 캐쉬의 실현은 이미 3장에서 고찰하였다.

파일체계의 올려태우기

파일체계가 올려태우기될 때(참조를 쉽게 하기 위하여 이 장의 마지막부분에 인쇄해 넣은 fs/super.c에 의하여 수행된다.) 실행되는 사건렬은 다음과 같다.

1. do_umount()
2. read_super()
3. ext2_read_super()(ext2파일체계의 경우)

이것들은 상위블로크를 귀환시킨다. 상위블로크는 우의 ext2_sops에서 볼수 있는 함수들에 대한 지적자구조체에 대한 지적자를 포함한다.

다른 중요한 자료를 포함하는데 그것들의 정의는 이 장의 마지막에 주었다.

파일에로의 접근

일단 파일체계가 제대로 올려태우기되면 거기서 파일로 접근할수 있다. 파일에로 접근하는데는 두개의 동작이 필요하다.

1. 어느 색인마디를 지적하는가를 찾기 위한 이름의 검색
2. 색인마디에로의 접근

VFS가 이름을 찾으면 그 이름에는 경로가 포함되여 있다(3장을 볼것). 따라서 파일이름이 절대적이 아닐 때는 경로가 "/"문자로 시작되며 그것은 체계호출이 진행되는 현행등록부와 관련되게 된다. 다음으로 핵심부는 정의된 파일체계안의 파일을 검색하기 위하여 체계정의코드를 리용하게 된다.

계속하여 경로이름의 요소를 얻고 그것을 검색한다. 만일 검색한 객체가 등록부이면 먼저 검색한 객체에 의하여 귀환된 등록부안에서 검색한다. 검색된 매 요소는 그것이 파일이든지 등록부이든지 관계없이 객체를 일의적으로 식별할수 있는 색인마디번호를 귀환하며 귀환된 그 내용에 의하여 접근할수 있다. 만일 파일이 다른 파일에 대한 기호적련결로 전환되면 VFS는 기호적련결로 검색되는 새로운 이름으로 시작한다. 무한재귀를 방지하기 위하여 기호련결의 깊이(depth)에 한계를 준다. 즉 핵심부는 내리기에 앞서 한행에서 몇개의 기호련결을 추적하게 된다.

namei()는 이름을 색인마디번호로 대응시키는데 리용된다. open_name()함수는 이 책의 마지막부분에 포함되여 있다.

일단 색인마디번호가 얻어 지면 그 파일에 접근할수 있다. Iget()함수는 색인마디번호에 의하여 정의되는 색인마디를 찾아서 귀환시킨다. Iput()함수는 색인마디에 대한 접근을 해제하기 위하여 그다음에 리용된다. 이 내용은 한개이상의 프로쎄스가 한번에 열리는 색인마디를 한개만 보관할수 있다는것을 제외하고는 malloc(), free()와 같으며 참조계수값은 그것이 비여 지거나 없어 지는 시각을 알아 내는데 리용된다.

응용프로그람코드로 되넘겨 지는 옹근수파일핸들은 그 프로쎄스에 해당한 파일표에로 접근하기 위한 변위값이다. 이 파일표슬로트는 파일이 닫겨 지거나 프로쎄스가 완료될 때까지 namei()함수로 검색된 색인마디번호를 보존한다. 프로쎄스는 파일핸들을 리용하는 "파일"에 대하여 그 어떤 일을 할 때마다 그 문제안에서 색인마디는 실제적으로 조종된다.

색인마디조작

앞에서 본바와 같이 파일에로의 접근은 그 파일의 색인마디에 대한 찾기를 의미한다. VFS가 파일체계에서 이름을 어떻게 찾으며 어떻게 색인마디를 돌려 주는가를 보자.

이 조작에서는 경로이름의 시작으로부터 출발하며 그 경로안의 첫번째 등록부의 색인마디를 검색한다. 다음에는 경로안의 다음등록부를 찾는데 그 색인마디를 리용한다. 만일 끝에 도달하면 검색하려는 파일이나 등록부의 색인마디가 발견된다. 조작을 계속하려면 색인마디가 필요한데 그러면 첫번째 검색으로부터 어떻게 이 조작을 계속할것인가?

파일체계에 관한 색인마디구조체를 지적하는 s_mounted로 호출된 상위블로크안에 보존되는 색인마디지적자가 있다. 이 색인마디는 파일체계가 올려태우기될 때 배정되며 올려태우기가 해제될 때 배정해제된다. 보통 ext2파일체계에서처럼 s_mounted 색인마디는 파일체계뿌리등록부의 색인마디이다. 거기로부터 시작하여 다른 모든 색인마디들을 검색할수 있다.

매 색인마디는 함수에 대한 구조체지적자를 포함한다. 그것이 바로 색인마디 inode_operations구조체이다.

이 구조체의 요소들중의 하나를 lookup()이라고 부르며 이것은 동일한 파일체계안에서 다른 색인마디들을 검색하는데 리용된다.

일반적으로 파일체계는 그 파일체계안의 모든 색인마디에 꼭 같은 한개의 lookup()함수만을 가지는데 이것으로 하여 여러개의 서로 다른 lookup()함수들을 가지는것과 파일체계에 적합한 객체로 그것을 지적할수 있다. proc파일체계는 파일체계안의 서로 다른 등록부들이 서로 다른 목적을 가지고 있기때문에 이 방법을 리용할수 있다.

Inode_operations구조체는 다음과 같다(<linux/fs.h>에서 볼수 있는 거의 모든것과 류사하게 정의되여 있다.).

```
struct inode_operations {
    struct file_operations* default_file_ops;
    int (*create) (struct inode*, const  char*, int, int, struct inode*);
    int (*lookup) (struct inode* , const  char*, int, struct inode**);
```

```
 int (*link) (struct inode*, struct inode*, const  char*, int);
 int (*unlink) (struct inode*, const  char*, int);
 int (*symlink) (struct inode*, const  char*, int, const char*);
 int (*mkdir) (struct inode*, const  char*, int, int);
 int (*rmdir) (struct inode*, const  char*, int);
 int (*mknod) (struct inode*, const  char*, int, int, int);
 int (*rename) (struct inode*, const  char*, int, struct
 inode, const char*, int);
 int (*realink) (struct inode*, char*, int);
 int (*follow_link) (struct inode*, struct inode*, int,
 int, struct inode**);
 int (*readpage) (struct inode*, struct page*);
 int (*writepage) (struct inode*, struct page *);
 int (*bmap) (struct inode*, int);
 void (*truncate) (struct inode *);
 int (*permission)(struct inode*, int);
 int (*smap)(struct inode*, int);
}
```

이 대다수의 기능들은 Linux체계호출로 직접적으로 넘겨 진다. ext2파일체계에서 등록부, 파일 그리고 기호련결들은 서로 다른 inode_operations를 가지고 있다. fs/ext2/dir.c파일은 ext2_dir_inode_operations를 포함하며 fs/ext2/file은 ext2_file_inode_operations를 포함하며 fs/ext2/symlink.c는 ext2_symlink_inode_ operations를 포함한다.

이 모든 파일들은 이 장의 마지막에 포함되여 있다.

inode_operations구조체에 밝혀 지지 않은 파일(등록부)과 관련된 체계호출도 많다. 그 체계호출에 대하여서는 file_operations구조체에 밝혀 져 있다. file_operations구조체는 장치구동기를 쓸 때 리용되는 구조체와 꼭 같으며 색인마디보다도 파일에 대하여 작업하는 조작들을 포함하고 있다.

```
struct file_operations{
   int (*lseek) (struct inode*, struct file*, off_t, int);
   int (*read) (struct inode*, struct file, char*, int);
   int (*write) (struct inode*, struct file*, const char*, int);
   int (*readdir) (struct inode*, struct file*, void*, filldir_t);
   int (*select) (struct inode*, struct file*, int, select_ table);
   int (*ioctl) (struct inode*, struct file*,unsigned int, nsigned long);
   int (*mmap) (struct inode*, struct file*, struct vm_area_struct);
   int (*open) (struct inode*, struct file*);
   void (*release) (struct inode*, struct file*);
   int (*fsync) (struct inode*, struct file*);
   int (*fasync) (struct inode*, struct file*, int);
   int (*check_media_change) (kdev_t dev);
```

```
    int (*revalidate) (kdev_t dev);
};
```

체계호출과 직접적으로 관련되지 않은 몇개의 함수들도 있으며 그 함수들은 리용되지는 않고 보통 NULL로 설정되여 있다.

원천파일 include/linux/fs.h(2.4.3)

```
#ifndef _Linux_FS_H
#define _Linux_FS_H
/*
 * This file has definitions for some important file table
 * structures etc.
 */
#include <linux/config.h>
#include <linux/linkage.h>
#include <linux/limits.h>
#include <linux/wait.h>
#include <linux/types.h>
#include <linux/vfs.h>
#include <linux/net.h>
#include <linux/kdev_t.h>
#include <linux/ioctl.h>
#include <linux/list.h>
#include <linux/dcache.h>
#include <linux/stat.h>
#include <linux/cache.h>
#include <linux/stddef.h>
#include <linux/string.h>

#include <asm/atomic.h>
#include <asm/bitops.h>
struct poll_table_struct;

/*
 * It's silly to have NR_OPEN bigger than NR_FILE, but you can change
 * the file limit at runtime and only root can increase the per-process
 * nr_file rlimit, so it's safe to set up a ridiculously high absolute
 * upper limit on files-per-process.
 *
 * Some programs (notably those using select()) may have to be
```

```
 * recompiled to take full advantage of the new limits..
 */
/* Fixed constants first: */
#undef NR_OPEN
#define NR_OPEN (1024*1024)    /* Absolute upper limit on fd num */
#define INR_OPEN 1024          /* Initial setting for nfile rlimits */

#define BLOCK_SIZE_BITS 10
#define BLOCK_SIZE (1<<BLOCK_SIZE_BITS)

/* And dynamically-tunable limits and defaults: */
struct files_stat_struct {
      int nr_files;            /* read only */
      int nr_free_files;       /* read only */
      int max_files;           /* tunable */
};
extern struct files_stat_struct files_stat;
extern int max_super_blocks, nr_super_blocks;
extern int leases_enable, dir_notify_enable, lease_break_time;

#define NR_FILE  8192        /* this can well be larger on a larger system */
#define NR_RESERVED_FILES 10 /* reserved for root */
#define NR_SUPER 256

#define MAY_EXEC 1
#define MAY_WRITE 2
#define MAY_READ 4

#define FMODE_READ 1
#define FMODE_WRITE 2

#define READ 0
#define WRITE 1
#define READA 2          /* read-ahead  - don't block if no resources */
#define SPECIAL 4  /* For non-blockdevice requests in request queue */

#define SEL_IN          1
#define SEL_OUT         2
#define SEL_EX          4
```

```
/* public flags for file_system_type */
#define FS_REQUIRES_DEV 1
#define FS_NO_DCACHE    2 /* Only dcache the necessary things. */
#define FS_NO_PRELIM    4 /* prevent preloading of dentries, even if
                           * FS_NO_DCACHE is not set.
                           */

#define FS_SINGLE 8 /*
                     * Filesystem that can have only one superblock;
                     * kernel-wide vfsmnt is placed in ->kern_mnt by
                     * kern_mount() which must be called _after_
                     * register_filesystem().
                     */

#define FS_NOMOUNT16 /* Never mount from userland */
#define FS_LITTER 32 /* Keeps the tree in dcache */
#define FS_ODD_RENAME   32768 /* Temporary stuff; will go away as soon
                               * as nfs_rename() will be cleaned up
                               */
/*
 * These are the fs-independent mount-flags: up to 32 flags are supported
 */
#define MS_RDONLY       1     /* Mount read-only */
#define MS_NOSUID       2     /* Ignore suid and sgid bits */
#define MS_NODEV        4     /* Disallow access to device special files */
#define MS_NOEXEC       8     /* Disallow program execution */
#define MS_SYNCHRONOUS  16    /* Writes are synced at once */
#define MS_REMOUNT      32    /* Alter flags of a mounted FS */
#define MS_MANDLOCK     64    /* Allow mandatory locks on an FS */
#define MS_NOATIME      1024  /* Do not update access times. */
#define MS_NODIRATIME   2048  /* Do not update directory access times */
#define MS_BIND         4096
/*
 * Flags that can be altered by MS_REMOUNT
 */
#define MS_RMT_MASK     (MS_RDONLY|MS_NOSUID|MS_NODEV|MS_NOEXEC|\
                  S_SYNCHRONOUS|MS_MANDLOCK|MS_NOATIME|MS_NODIRATIME)
```

```
/*
 * Magic mount flag number. Has to be or-ed to the flag values.
 */
#define MS_MGC_VAL 0xC0ED0000   /* magic flag number to indicate "new" flags */
#define MS_MGC_MSK 0xffff0000  /* magic flag number mask */

/* Inode flags - they have nothing to superblock flags now */

#define S_SYNC          1      /* Writes are synced at once */
#define S_NOATIME       2      /* Do not update access times */
#define S_QUOTA         4      /* Quota initialized for file */
#define S_APPEND        8      /* Append-only file */
#define S_IMMUTABLE     16     /* Immutable file */
#define S_DEAD          32     /* removed, but still open directory */

/*
 * Note that nosuid etc flags are inode-specific: setting some file-system
 * flags just means all the inodes inherit those flags by default. It might be
 * possible to override it selectively if you really wanted to with some
 * ioctl() that is not currently implemented.
 *
 * Exception: MS_RDONLY is always applied to the entire file system.
 *
 * Unfortunately, it is possible to change a filesystems flags with it mounted
 * with files in use.  This means that all of the inodes will not have their
 * i_flags updated.  Hence, i_flags no longer inherit the superblock mount
 * flags, so these have to be checked separately. -- rmk@arm.uk.linux.org
 */
#define __IS_FLG(inode,flg) ((inode)->i_sb->s_flags & (flg))

#define IS_RDONLY(inode) ((inode)->i_sb->s_flags & MS_RDONLY)
#define IS_NOSUID(inode)	__IS_FLG(inode, MS_NOSUID)
#define IS_NODEV(inode)		__IS_FLG(inode, MS_NODEV)
#define IS_NOEXEC(inode)	__IS_FLG(inode, MS_NOEXEC)
#define IS_SYNC(inode)		(__IS_FLG(inode, MS_SYNCHRONOUS) || ((inode)->i_flags & S_SYNC))
#define IS_MANDLOCK(inode)	__IS_FLG(inode, MS_MANDLOCK)

#define IS_QUOTAINIT(inode)	((inode)->i_flags & S_QUOTA)
```

```
#define IS_APPEND(inode) ((inode)->i_flags & S_APPEND)
#define IS_IMMUTABLE(inode)   ((inode)->i_flags & S_IMMUTABLE)
#define IS_NOATIME(inode)     (__IS_FLG(inode, MS_NOATIME) || ((inode)-
                              >i_flags & S_NOATIME))
#define IS_NODIRATIME(inode)  __IS_FLG(inode, MS_NODIRATIME)

#define IS_DEADDIR(inode)     ((inode)->i_flags & S_DEAD)

/* the read-only stuff doesn't really belong here, but any other place is
   probably as bad and I don't want to create yet another include file. */

#define BLKROSET   _IO(0x12,93)      /* set device read-only (0 = read-write) */
#define BLKROGET   _IO(0x12,94)      /* get read-only status (0 = ead_write) */
#define BLKRRPART  _IO(0x12,95)      /* re-read partition table */
#define BLKGETSIZE _IO(0x12,96)      /* return device size */
#define BLKFLSBUF  _IO(0x12,97)      /* flush buffer cache */
#define BLKRASET   _IO(0x12,98)      /* Set read ahead for block device */
#define BLKRAGET   _IO(0x12,99)      /* get current read ahead setting */
#define BLKFRASET  _IO(0x12,100)     /* set filesystem (mm/filemap.c)
                                        read-ahead */
#define BLKFRAGET  _IO(0x12,101)     /* get filesystem (mm/filemap.c) read-
                                        ahead */
#define BLKSECTSET _IO(0x12,102)     /* set max sectors per request
                                        (ll_rw_blk.c) */
#define BLKSECTGET _IO(0x12,103)     /* get max sectors per request
                                        (ll_rw_blk.c) */
#define BLKSSZGET  _IO(0x12,104)     /* get block device sector size */
#if 0
#define BLKPG      _IO(0x12,105)     /* See blkpg.h */
#define BLKELVGET  _IOR(0x12,106,sizeof(blkelv_ioctl_arg_t))/* elevator get */
#define BLKELVSET  _IOW(0x12,107,sizeof(blkelv_ioctl_arg_t))/* elevator set */
/* This was here just to show that the number is taken -
   probably all these _IO(0x12,*) ioctls should be moved to blkpg.h. */
#endif

#define BMAP_IOCTL 1                /* obsolete - kept for compatibility */
#define FIBMAP        _IO(0x00,1)   /* bmap access */
#define FIGETBSZ   _IO(0x00,2)      /* get the block size used for bmap */
```

```
#ifdef __KERNEL__

#include <asm/semaphore.h>
#include <asm/byteorder.h>

extern void update_atime (struct inode *);
#define UPDATE_ATIME(inode) update_atime (inode)

extern void buffer_init(unsigned long);
extern void inode_init(unsigned long);

/* bh state bits */
#define BH_Uptodate     0     /* 1 if the buffer contains valid data */
#define BH_Dirty        1     /* 1 if the buffer is dirty */
#define BH_Lock         2     /* 1 if the buffer is locked */
#define BH_Req          3     /* 0 if the buffer has been invalidated */
#define BH_Mapped       4     /* 1 if the buffer has a disk mapping */
#define BH_New          5     /* 1 if the buffer is new and not yet written out */
#define BH_Protected    6     /* 1 if the buffer is protected */

/*
 * Try to keep the most commonly used fields in single cache lines (16
 * bytes) to improve performance. This ordering should be
 * particularly beneficial on 32-bit processors.
 *
 * We use the first 16 bytes for the data which is used in searches
 * over the block hash lists (ie. getblk() and friends).
 *
 * The second 16 bytes we use for lru buffer scans, as used by
 * sync_buffers() and refill_freelist().  -- sct
 */
struct buffer_head {
      /* First cache line:*/
      struct buffer_head *b_next;   /* Hash queue list */
      unsigned long b_blocknr;      /* block number */
      unsigned short b_size;        /* block size */
      unsigned short b_list;        /* List that this buffer appears */
      kdev_t b_dev;                 /* device (B_FREE = free) */
```

```
	atomic_t b_count;		/* users using this block */
	kdev_t b_rdev;			/* Real device */
	unsigned long b_state;		/* buffer state bitmap (see above) */
	unsigned long b_flushtime;	/* Time when (dirty) buffer
					   should be written */

	struct buffer_head *b_next_free; /* lru/free list linkage */
	struct buffer_head *b_prev_free; /* doubly linked list of buffers */
	struct buffer_head *b_this_page; /* circular list of buffers in one
						page */
	struct buffer_head *b_reqnext;	/* request queue */

	struct buffer_head **b_pprev; /* doubly linked list of hash-queue */
	char * b_data;			/* pointer to data block (512 byte) */
	struct page *b_page;		/* the page this bh is mapped to */
		void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O
	completion */
	void *b_private;		/* reserved for b_end_io */

	unsigned long b_rsector; /* Real buffer location on disk */
	wait_queue_head_t b_wait;

	struct inode *		b_inode;
	struct list_head	b_inode_buffers;	/* doubly linked list of
							   inode dirty buffers */
};

typedef void (bh_end_io_t)(struct buffer_head *bh, int uptodate);
void init_buffer(struct buffer_head *, bh_end_io_t *, void *);

#define __buffer_state(bh, state) (((bh)->b_state & (1UL << BH_##state)) != 0)

#define buffer_uptodate(bh)	__buffer_state(bh,Uptodate)
#define buffer_dirty(bh)	__buffer_state(bh,Dirty)
#define buffer_locked(bh)	__buffer_state(bh,Lock)
#define buffer_req(bh)		__buffer_state(bh,Req)
#define buffer_mapped(bh)	__buffer_state(bh,Mapped)
#define buffer_new(bh)		__buffer_state(bh,New)
#define buffer_protected(bh)	__buffer_state(bh,Protected)
```

```
#define bh_offset(bh)          ((unsigned long)(bh)->b_data & PAGE_MASK)
extern void set_bh_page(struct buffer_head *bh, struct page *page, unsigned
long offset);

#define touch_buffer(bh) SetPageReferenced(bh->b_page)

#include <linux/pipe_fs_i.h>
#include <linux/minix_fs_i.h>
#include <linux/ext2_fs_i.h>
#include <linux/hpfs_fs_i.h>
#include <linux/ntfs_fs_i.h>
#include <linux/msdos_fs_i.h>
#include <linux/umsdos_fs_i.h>
#include <linux/iso_fs_i.h>
#include <linux/nfs_fs_i.h>
#include <linux/sysv_fs_i.h>
#include <linux/affs_fs_i.h>
#include <linux/ufs_fs_i.h>
#include <linux/efs_fs_i.h>
#include <linux/coda_fs_i.h>
#include <linux/romfs_fs_i.h>
#include <linux/shmem_fs.h>
#include <linux/smb_fs_i.h>
#include <linux/hfs_fs_i.h>
#include <linux/adfs_fs_i.h>
#include <linux/qnx4_fs_i.h>
#include <linux/bfs_fs_i.h>
#include <linux/udf_fs_i.h>
#include <linux/ncp_fs_i.h>
#include <linux/proc_fs_i.h>
#include <linux/usbdev_fs_i.h>
#include <linux/jfs_fs_i.h>
/*
 * Attribute flags.  These should be or-ed together to figure out what
 * has been changed!
 */

#define ATTR_MODE       1
#define ATTR_UID        2
```

```
#define ATTR_GID        4
#define ATTR_SIZE       8
#define ATTR_ATIME      16
#define ATTR_MTIME      32
#define ATTR_CTIME      64
#define ATTR_ATIME_SET  128
#define ATTR_MTIME_SET  256
#define ATTR_FORCE      512   /* Not a change, but a change it */
#define ATTR_ATTR_FLAG  1024

/*
 * This is the Inode Attributes structure, used for notify_change().  It
 * uses the above definitions as flags, to know which values have changed.
 * Also, in this manner, a Filesystem can look at only the values it cares
 * about.  Basically, these are the attributes that the VFS layer can
 * request to change from the FS layer.
 *
 * Derek Atkins <warlord@MIT.EDU> 94-10-20
 */
struct iattr {
      unsigned int ia_valid;
      umode_t           ia_mode;
      uid_t             ia_uid;
      gid_t             ia_gid;
      loff_t            ia_size;
      time_t            ia_atime;
      time_t            ia_mtime;
      time_t            ia_ctime;
      unsigned int ia_attr_flags;
};

/*
 * This is the inode attributes flag definitions
 */

#define ATTR_FLAG_SYNCRONOUS   1     /* Syncronous write */
#define ATTR_FLAG_NOATIME      2     /* Don't update atime */
#define ATTR_FLAG_APPEND       4     /* Append-only file */
#define ATTR_FLAG_IMMUTABLE    8     /* Immutable file */
```

```
#define ATTR_FLAG_NODIRATIME   16    /* Don't update atime for directory */

/*
 * Includes for diskquotas and mount structures.
 */
#include <linux/quota.h>
#include <linux/mount.h>
/*
 * oh the beauties of C type declarations.
 */
struct page;
struct address_space;

struct address_space_operations {
      int (*writepage)(struct page *);
      int (*readpage)(struct file *, struct page *);
      int (*sync_page)(struct page *);
      int (*prepare_write)(struct file *, struct page *, unsigned,
      unsigned);
      int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
      /* Unfortunately this kludge is needed for FIBMAP. Don't use it */
      int (*bmap)(struct address_space *, long);
};

struct address_space {
      struct list_head   clean_pages;      /* list of clean pages */
      struct list_head   dirty_pages;      /* list of dirty pages */
      struct list_head   locked_pages;     /* list of locked pages */
      unsigned long            nrpages;    /* number of total pages */
      struct address_space_operations *a_ops;  /* methods */
      struct inode *host;                  /* owner: inode, block_device */
      struct vm_area_struct    *i_mmap;    /* list of private mappings */
      struct vm_area_struct    *i_mmap_shared; /* list of shared mappings */
      spinlock_t  i_shared_lock;  /* and spinlock protecting it */
};

struct block_device {
      struct list_head   bd_hash;
      atomic_t           bd_count;
```

```
/*      struct address_space     bd_data; */
        dev_t         bd_dev;  /* not a kdev_t - it's a search key */
        atomic_t            bd_openers;
        const struct block_device_operations *bd_op;
        struct semaphore   bd_sem;     /* open/close mutex */
};

struct inode {
        struct list_head         i_hash;
        struct list_head         i_list;
        struct list_head         i_dentry;

        struct list_head         i_dirty_buffers;

        unsigned long            i_ino;
        atomic_t                 i_count;
        kdev_t                   i_dev;
        umode_t                  i_mode;
        nlink_t                  i_nlink;
        uid_t                    i_uid;
        gid_t                    i_gid;
        kdev_t                   i_rdev;
        loff_t                   i_size;
        time_t                   i_atime;
        time_t                   i_mtime;
        time_t                   i_ctime;
        unsigned long            i_blksize;
        unsigned long            i_blocks;
        unsigned long            i_version;
        struct semaphore         i_sem;
        struct semaphore         i_zombie;
        struct inode_operations  *i_op;
        struct file_operations *i_fop;     /* former ->i_op-
        >default_file_ops */
        struct super_block *i_sb;
        wait_queue_head_t        i_wait;
        struct file_lock         *i_flock;
        struct address_space     *i_mapping;
        struct address_space     i_data;
```

```
struct dquot             *i_dquot[MAXQUOTAS];
struct pipe_inode_info   *i_pipe;
struct block_device      *i_bdev;

unsigned long      i_dnotify_mask; /* Directory notify events */
struct dnotify_struct    *i_dnotify; /* for directory notifications */

unsigned long            i_state;

unsigned int             i_flags;
unsigned char            i_sock;

atomic_t                 i_writecount;
unsigned int             i_attr_flags;
__u32                    i_generation;
union {
      struct minix_inode_info       minix_i;
      struct ext2_inode_info        ext2_i;
      struct hpfs_inode_info        hpfs_i;
      struct ntfs_inode_info        ntfs_i;
      struct msdos_inode_info       msdos_i;
      struct umsdos_inode_info umsdos_i;
      struct iso_inode_info         isofs_i;
      struct nfs_inode_info         nfs_i;
      struct sysv_inode_info        sysv_i;
      struct affs_inode_info        affs_i;
      struct ufs_inode_info         ufs_i;
      struct efs_inode_info         efs_i;
      struct romfs_inode_info       romfs_i;
      struct shmem_inode_info       shmem_i;
      struct coda_inode_info        coda_i;
      struct smb_inode_info         smbfs_i;
      struct hfs_inode_info         hfs_i;
      struct adfs_inode_info        adfs_i;
      struct qnx4_inode_info        qnx4_i;
      struct bfs_inode_info         bfs_i;
      struct udf_inode_info         udf_i;
      struct ncp_inode_info         ncpfs_i;
      struct proc_inode_info        proc_i;
```

```
		struct socket			socket_i;
		struct usbdev_inode_info	usbdev_i;
		struct jfs_inode_info		jfs_i;
		void				*generic_ip;
	} u;
};

struct fown_struct {
	int pid;		/* pid or -pgrp where SIGIO should be sent */
	uid_t uid, euid;	/* uid/euid of process setting the owner */
	int signum;		/* posix.1b rt signal to be delivered on IO */
};

struct file {
	struct list_head	f_list;
	struct dentry		*f_dentry;
	struct vfsmount		*f_vfsmnt;
	struct file_operations	*f_op;
	atomic_t		f_count;
	unsigned int		f_flags;
	mode_t			f_mode;
	loff_t			f_pos;
	unsigned long		f_reada, f_ramax, f_raend, f_ralen, f_rawin;
	struct fown_struct	f_owner;
	unsigned int		f_uid, f_gid;
	int			f_error;

	unsigned long		f_version;

	/* needed for tty driver, and maybe others */
	void			*private_data;
};
extern spinlock_t files_lock;
#define file_list_lock() spin_lock(&files_lock);
#define file_list_unlock() spin_unlock(&files_lock);

#define get_file(x)	atomic_inc(&(x)->f_count)
#define file_count(x)	atomic_read(&(x)->f_count)

extern int init_private_file(struct file *, struct dentry *, int);
```

```
#define     MAX_NON_LFS ((1UL<<31) - 1)

#define FL_POSIX 1
#define FL_FLOCK 2
#define FL_BROKEN 4    /* broken flock() emulation */
#define FL_ACCESS 8    /* for processes suspended by mandatory locking */
#define FL_LOCKD 16    /* lock held by rpc.lockd */
#define FL_LEASE 32    /* lease held on this file */

/*
 * The POSIX file lock owner is determined by
 * the "struct files_struct" in the thread group
 * (or NULL for no owner - BSD locks).
 *
 * Lockd stuffs a "host" pointer into this.
 */
typedef struct files_struct *fl_owner_t;

struct file_lock {
      struct file_lock *fl_next;   /* singly linked list for this inode
*/
      struct list_head fl_link;    /* doubly linked list of all locks */
      struct list_head fl_block;   /* circular list of blocked processes
*/
      fl_owner_t fl_owner;
      unsigned int fl_pid;
      wait_queue_head_t fl_wait;
      struct file *fl_file;
      unsigned char fl_flags;
      unsigned char fl_type;
      loff_t fl_start;
      loff_t fl_end;

      void (*fl_notify)(struct file_lock *);  /* unblock callback */
      void (*fl_insert)(struct file_lock *);      /* lock insertion callback */
      void (*fl_remove)(struct file_lock *);    /* lock removal callback */

      struct fasync_struct *   fl_fasync; /* for lease break notifications */
```

```
    union {
          struct nfs_lock_info   nfs_fl;
    } fl_u;
};

/* The following constant reflects the upper bound of the file/locking
space */
#ifndef OFFSET_MAX
#define INT_LIMIT(x)   (~((x)1 << (sizeof(x)*8 - 1)))
#define OFFSET_MAX        INT_LIMIT(loff_t)
#define OFFT_OFFSET_MAX   INT_LIMIT(off_t)
#endif

extern struct list_head file_lock_list;

#include <linux/fcntl.h>

extern int fcntl_getlk(unsigned int, struct flock *);
extern int fcntl_setlk(unsigned int, unsigned int, struct flock *);
extern asmlinkage long sys_fcntl(unsigned int fd, unsigned int cmd,
unsigned long arg);
extern asmlinkage long sys_dup(unsigned int fildes);
extern asmlinkage long sys_dup2(unsigned int oldfd, unsigned int newfd);
extern asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t
                                    count);
extern asmlinkage ssize_t sys_write(unsigned int fd, const char * buf,
                                    size_t count);
extern asmlinkage long sys_chroot(const char * filename);
extern asmlinkage long sys_chdir(const char * filename);


extern int fcntl_getlk64(unsigned int, struct flock64 *);
extern int fcntl_setlk64(unsigned int, unsigned int, struct flock64 *);

/* fs/locks.c */
extern void locks_init_lock(struct file_lock *);
extern void locks_copy_lock(struct file_lock *, struct file_lock *);
extern void locks_remove_posix(struct file *, fl_owner_t);
extern void locks_remove_flock(struct file *);
extern struct file_lock *posix_test_lock(struct file *, struct
```

```
                                        file_lock *);
extern int posix_lock_file(struct file *, struct file_lock *, unsigned int);
extern void posix_block_lock(struct file_lock *, struct file_lock *);
extern int posix_locks_deadlock(struct file_lock *, struct file_lock *);
extern void posix_unblock_lock(struct file_lock *);
extern int __get_lease(struct inode *inode, unsigned int flags);
extern time_t lease_get_mtime(struct inode *);
extern int lock_may_read(struct inode *, loff_t start, unsigned long count);
extern int lock_may_write(struct inode *, loff_t start, unsigned long count);

struct fasync_struct {
      int   magic;
      int   fa_fd;
      struct      fasync_struct    *fa_next; /* singly linked list */
      struct      file             *fa_file;
};

#define FASYNC_MAGIC 0x4601

/* SMP safe fasync helpers: */
extern int fasync_helper(int, struct file *, int, struct fasync_struct **);
/* can be called from interrupts */
extern void kill_fasync(struct fasync_struct **, int, int);
/* only for net: no internal synchronization */
extern void __kill_fasync(struct fasync_struct *, int, int);

struct nameidata {
      struct dentry *dentry;
      struct vfsmount *mnt;
      struct qstr last;
      unsigned int flags;
      int last_type;
};

#define DQUOT_USR_ENABLED    0x01        /* User diskquotas enabled */
#define DQUOT_GRP_ENABLED    0x02        /* Group diskquotas enabled */

struct quota_info
{
      unsigned int flags;                /* Flags for diskquotas on this
```

```
                                                device */
      struct semaphore dqio_sem;        /* lock device while I/O in
                                           progress */
      struct semaphore dqoff_sem;       /* serialize quota_off() and
                                           quota_on() on device */
      struct file *files[MAXQUOTAS];    /* fp's to quotafiles */
      struct mem_dqinfo info[MAXQUOTAS];/* Information for each
                                             quota type */
};

/*
 *    Umount options
 */

#define MNT_FORCE 0x00000001  /* Attempt to forcibily umount */

#include <linux/minix_fs_sb.h>
#include <linux/ext2_fs_sb.h>
#include <linux/hpfs_fs_sb.h>
#include <linux/ntfs_fs_sb.h>
#include <linux/msdos_fs_sb.h>
#include <linux/iso_fs_sb.h>
#include <linux/nfs_fs_sb.h>
#include <linux/sysv_fs_sb.h>
#include <linux/affs_fs_sb.h>
#include <linux/ufs_fs_sb.h>
#include <linux/efs_fs_sb.h>
#include <linux/romfs_fs_sb.h>
#include <linux/smb_fs_sb.h>
#include <linux/hfs_fs_sb.h>
#include <linux/adfs_fs_sb.h>
#include <linux/qnx4_fs_sb.h>
#include <linux/reiserfs_fs_sb.h>
#include <linux/bfs_fs_sb.h>
#include <linux/udf_fs_sb.h>
#include <linux/ncp_fs_sb.h>
#include <linux/usbdev_fs_sb.h>

extern struct list_head super_blocks;
```

```
#define sb_entry(list) list_entry((list), struct super_block, s_list)
struct super_block {
      struct list_head s_list;    /* Keep this first */
      kdev_t                      s_dev;
      unsigned long               s_blocksize;
      unsigned char               s_blocksize_bits;
      unsigned char               s_lock;
      unsigned char               s_dirt;
      unsigned long long          s_maxbytes; /* Max file size */
      struct file_system_type     *s_type;
      struct super_operations     *s_op;
      struct dquot_operations     *dq_op;
      unsigned long               s_flags;
      unsigned long               s_magic;
      struct dentry               *s_root;
      wait_queue_head_t           s_wait;

      struct list_head s_dirty;   /* dirty inodes */
      struct list_head s_files;

      struct block_device    *s_bdev;
      struct list_head s_mounts;  /* vfsmount(s) of this one */
      struct quota_infos_dquot;   /* Diskquota specific options */

      union {
            struct minix_sb_info  minix_sb;
            struct ext2_sb_info   ext2_sb;
            struct hpfs_sb_info   hpfs_sb;
            struct ntfs_sb_info   ntfs_sb;
            struct msdos_sb_info  msdos_sb;
            struct isofs_sb_info  isofs_sb;
            struct nfs_sb_info    nfs_sb;
            struct sysv_sb_info   sysv_sb;
            struct affs_sb_info   affs_sb;
            struct ufs_sb_info    ufs_sb;
            struct efs_sb_info    efs_sb;
            struct shmem_sb_info  shmem_sb;
            struct romfs_sb_info  romfs_sb;
            struct smb_sb_info    smbfs_sb;
            struct hfs_sb_info    hfs_sb;
```

```
		struct adfs_sb_info	adfs_sb;
		struct qnx4_sb_info	qnx4_sb;
		struct reiserfs_sb_info reiserfs_sb;
		struct bfs_sb_info	bfs_sb;
		struct udf_sb_info	udf_sb;
		struct ncp_sb_info	ncpfs_sb;
		struct usbdev_sb_info   usbdevfs_sb;
		void			*generic_sbp;
	} u;
	/*
	 * The next field is for VFS *only*. No filesystems have any business
	 * even looking at it. You had been warned.
	 */
	struct semaphore s_vfs_rename_sem;	/* Kludge */

	/* The next field is used by knfsd when converting a (inode number based)
	 * file handle into a dentry. As it builds a path in the dcache tree from
	 * the bottom up, there may for a time be a subpath of dentrys which is not
	 * connected to the main tree.  This semaphore ensure that there is only ever
	 * one such free path per filesystem.  Note that unconnected files (or other
	 * non-directories) are allowed, but not unconnected directories.
	 */
	struct semaphore s_nfsd_free_path_sem;
};

/*
 * VFS helper functions..
 */
extern int vfs_create(struct inode *, struct dentry *, int);
extern int vfs_mkdir(struct inode *, struct dentry *, int);
extern int vfs_mknod(struct inode *, struct dentry *, int, dev_t);
extern int vfs_symlink(struct inode *, struct dentry *, const char *);
extern int vfs_link(struct dentry *, struct inode *, struct dentry *);
extern int vfs_rmdir(struct inode *, struct dentry *);
extern int vfs_unlink(struct inode *, struct dentry *);
```

```
extern int vfs_rename(struct inode *, struct dentry *, struct inode *,
struct dentry *);

/*
 * File types
 */
#define DT_UNKNOWN     0
#define DT_FIFO        1
#define DT_CHR         2
#define DT_DIR         4
#define DT_BLK         6
#define DT_REG         8
#define DT_LNK         10
#define DT_SOCK        12
#define DT_WHT         14

/*
 * This is the "filldir" function type, used by readdir() to let
 * the kernel specify what kind of direct layout it wants to have.
 * This allows the kernel to read directories into kernel space or
 * to have different direct layouts depending on the binary type.
 */
typedef int (*filldir_t)(void *, const char *, int, off_t, ino_t,
unsigned);

struct block_device_operations {
      int (*open) (struct inode *,    struct file *);
      int (*release) (struct inode *, struct file *);
      int (*ioctl) (struct inode *,   struct file *, unsigned, unsigned
long);
      int (*check_media_change) (kdev_t);
      int (*revalidate) (kdev_t);
};

/*
 * NOTE:
 * read, write, poll, fsync, readv, writev can be called
 *   without the big kernel lock held in all filesystems.
 */
struct file_operations {
```

```
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
            long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned
            long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned
            long, loff_t *);
        ssize_t (*writepage) (struct file *, struct page *, int, size_t,
            loff_t *, int);
};

struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int);
        struct dentry * (*lookup) (struct inode *,struct dentry *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,int);
        int (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *, int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*revalidate) (struct dentry *);
        int (*setattr) (struct dentry *, struct iattr *);
```

```
	int (*getattr) (struct dentry *, struct iattr *);
};

/*
 * NOTE: write_inode, delete_inode, clear_inode, put_inode can be called
 * without the big kernel lock held in all filesystems.
 */
struct super_operations {
	void (*read_inode) (struct inode *);

	/* reiserfs kludge.  reiserfs needs 64 bits of information to
	** find an inode.  We are using the read_inode2 call to get
	** that information.  We don't like this, and are waiting on some
	** VFS changes for the real solution.
	** iget4 calls read_inode2, iff it is defined
	*/
	void (*read_inode2) (struct inode *, void *);
	void (*dirty_inode) (struct inode *);
	void (*write_inode) (struct inode *, int);
	void (*put_inode) (struct inode *);
	void (*delete_inode) (struct inode *);
	void (*put_super) (struct super_block *);
	void (*write_super) (struct super_block *);
	void (*write_super_lockfs) (struct super_block *);
	void (*unlockfs) (struct super_block *);
	int (*statfs) (struct super_block *, struct statfs *);
	int (*remount_fs) (struct super_block *, int *, char *);
	void (*clear_inode) (struct inode *);
	void (*umount_begin) (struct super_block *);
};

/* Inode state bits.. */
#define I_DIRTY_SYNC		1 /* Not dirty enough for O_DATASYNC */
#define I_DIRTY_DATASYNC	2 /* Data-related inode changes pending */
#define I_DIRTY_PAGES		4 /* Data-related inode changes pending */
#define I_LOCK			8
#define I_FREEING		16
#define I_CLEAR			32

#define I_DIRTY (I_DIRTY_SYNC | I_DIRTY_DATASYNC | I_DIRTY_PAGES)
```

```
extern void __mark_inode_dirty(struct inode *, int);
static inline void mark_inode_dirty(struct inode *inode)
{
	if ((inode->i_state & I_DIRTY) != I_DIRTY)
		__mark_inode_dirty(inode, I_DIRTY);
}

static inline void mark_inode_dirty_sync(struct inode *inode)
{
	if (!(inode->i_state & I_DIRTY_SYNC))
		__mark_inode_dirty(inode, I_DIRTY_SYNC);
}

static inline void mark_inode_dirty_pages(struct inode *inode)
{
	if (inode && !(inode->i_state & I_DIRTY_PAGES))
		__mark_inode_dirty(inode, I_DIRTY_PAGES);
}

struct fown_struct {
	int pid;		/* pid or -pgrp where SIGIO should be sent */
	uid_t uid, euid;	/* uid/euid of process setting the owner */
	int signum;		/* posix.1b rt signal to be delivered on IO */
};
struct file {
	struct list_head		f_list;
	struct dentry			*f_dentry;
	struct vfsmount			*f_vfsmnt;
	struct file_operations	*f_op;
	atomic_t			f_count;
	unsigned int			f_flags;
	mode_t				f_mode;
	loff_t				f_pos;
	unsigned long			f_reada, f_ramax, f_raend, f_ralen, f_rawin;
	struct fown_struct		f_owner;
	unsigned int			f_uid, f_gid;
	int				f_error;
```

```
	unsigned long		f_version;

	/* needed for tty driver, and maybe others */
	void			*private_data;
};
extern spinlock_t files_lock;
#define file_list_lock() spin_lock(&files_lock);
#define file_list_unlock() spin_unlock(&files_lock);

#define get_file(x)	atomic_inc(&(x)->f_count)
#define file_count(x)	atomic_read(&(x)->f_count)

extern int init_private_file(struct file *, struct dentry *, int);

#define FL_POSIX	1
#define FL_FLOCK	2
#define FL_BROKEN	4	/* broken flock() emulation */
#define FL_ACCESS	8	/* for processes suspended by mandatory locking */
#define FL_LOCKD	16	/* lock held by rpc.lockd */
#define FL_LEASE	32	/* lease held on this file */

/*
 * The POSIX file lock owner is determined by
 * the "struct files_struct" in the thread group
 * (or NULL for no owner - BSD locks).
 *
 * Lockd stuffs a "host" pointer into this.
 */
typedef struct files_struct *fl_owner_t;

struct file_lock {
	struct file_lock *fl_next;	/* singly linked list for this inode  */
	struct list_head fl_link;	/* doubly linked list of all locks */
	struct list_head fl_block;	/* circular list of blocked processes */
	fl_owner_t fl_owner;
	unsigned int fl_pid;
	wait_queue_head_t fl_wait;
	struct file *fl_file;
	unsigned char fl_flags;
```

```
	unsigned char fl_type;
	loff_t fl_start;
	loff_t fl_end;

	void (*fl_notify)(struct file_lock *);	/* unblock callback */
	void (*fl_insert)(struct file_lock *);	/* lock insertion callback
	*/
	void (*fl_remove)(struct file_lock *);	/* lock removal callback */

	struct fasync_struct *	fl_fasync; /* for lease break notifications */

	union {
		struct nfs_lock_info	nfs_fl;
	} fl_u;
};

	/* The following constant reflects the upper bound of the file/locking
	space */
#ifndef OFFSET_MAX
#define INT_LIMIT(x)	(~((x)1 << (sizeof(x)*8 - 1)))
#define OFFSET_MAX	INT_LIMIT(loff_t)
#define OFFT_OFFSET_MAX	INT_LIMIT(off_t)
#endif

extern struct list_head file_lock_list;

#include <linux/fcntl.h>

extern int fcntl_getlk(unsigned int, struct flock *);
extern int fcntl_setlk(unsigned int, unsigned int, struct flock *);

extern int fcntl_getlk64(unsigned int, struct flock64 *);
extern int fcntl_setlk64(unsigned int, unsigned int, struct flock64 *);

/* fs/locks.c */
extern void locks_init_lock(struct file_lock *);
extern void locks_copy_lock(struct file_lock *, struct file_lock *);
extern void locks_remove_posix(struct file *, fl_owner_t);
extern void locks_remove_flock(struct file *);
```

```
extern struct file_lock *posix_test_lock(struct file *, struct file_lock *);
extern int posix_lock_file(struct file *, struct file_lock *, unsigned int);
extern void posix_block_lock(struct file_lock *, struct file_lock *);
extern void posix_unblock_lock(struct file_lock *);
extern int __get_lease(struct inode *inode, unsigned int flags);
extern time_t lease_get_mtime(struct inode *);
extern int lock_may_read(struct inode *, loff_t start, unsigned long count);
extern int lock_may_write(struct inode *, loff_t start, unsigned long count);
struct fasync_struct {
	int	magic;
	int	fa_fd;
	structfasync_struct	*fa_next; /* singly linked list */
	struct	file	*fa_file;
};

#define FASYNC_MAGIC 0x4601

/* SMP safe fasync helpers: */
extern int fasync_helper(int, struct file *, int, struct fasync_struct **);
/* can be called from interrupts */
extern void kill_fasync(struct fasync_struct **, int, int);
/* only for net: no internal synchronization */
extern void __kill_fasync(struct fasync_struct *, int, int);

struct nameidata {
	struct dentry *dentry;
	struct vfsmount *mnt;
	struct qstr last;
	unsigned int flags;
	int last_type;
};

#define DQUOT_USR_ENABLED	0x01	/* User diskquotas enabled */
#define DQUOT_GRP_ENABLED	0x02	/* Group diskquotas enabled */

struct quota_mount_options
{
	unsigned int flags;		/* Flags for diskquotas on this device */
	struct semaphore dqio_sem;	/* lock device while I/O in progress */
```

```
        struct semaphore dqoff_sem;    /* serialize quota_off() and quota_on()
        on device */
        struct file *files[MAXQUOTAS];       /* fp's to quotafiles */
        time_t inode_expire[MAXQUOTAS];      /* expiretime for inode-quota */
        time_t block_expire[MAXQUOTAS];      /* expiretime for block-quota */
        char rsquash[MAXQUOTAS];             /* for quotas threat root as any
        other user */
};

/*
 *      Umount options
 */

#define MNT_FORCE 0x00000001  /* Attempt to forcibily umount */

#include <linux/minix_fs_sb.h>
#include <linux/ext2_fs_sb.h>
#include <linux/hpfs_fs_sb.h>
#include <linux/ntfs_fs_sb.h>
#include <linux/msdos_fs_sb.h>
#include <linux/iso_fs_sb.h>
#include <linux/nfs_fs_sb.h>
#include <linux/sysv_fs_sb.h>
#include <linux/affs_fs_sb.h>
#include <linux/ufs_fs_sb.h>
#include <linux/efs_fs_sb.h>
#include <linux/romfs_fs_sb.h>
#include <linux/smb_fs_sb.h>
#include <linux/hfs_fs_sb.h>
#include <linux/adfs_fs_sb.h>
#include <linux/qnx4_fs_sb.h>
#include <linux/bfs_fs_sb.h>
#include <linux/udf_fs_sb.h>
#include <linux/ncp_fs_sb.h>
#include <linux/usbdev_fs_sb.h>
#include <linux/jfs_fs_sb.h>

extern struct list_head super_blocks;
```

```
#define sb_entry(list)  list_entry((list), struct super_block, s_list)
struct super_block {
      struct list_head        s_list;     /* Keep this first */
      kdev_t                  s_dev;
      unsigned long           s_blocksize;
      unsigned char           s_blocksize_bits;
      unsigned char           s_lock;
      unsigned char           s_dirt;
      struct file_system_type *s_type;
      struct super_operations *s_op;
      struct dquot_operations *dq_op;
      unsigned long           s_flags;
      unsigned long           s_magic;
      struct dentry           *s_root;
      wait_queue_head_t       s_wait;

      struct list_head        s_dirty;    /* dirty inodes */
      struct list_head        s_files;

      struct block_device     *s_bdev;
      struct list_head        s_mounts;   /* vfsmount(s) of this one */
      struct quota_mount_options s_dquot; /* Diskquota specific options */

      union {
            struct minix_sb_info    minix_sb;
            struct ext2_sb_info     ext2_sb;
            struct hpfs_sb_info     hpfs_sb;
            struct ntfs_sb_info     ntfs_sb;
            struct msdos_sb_info    msdos_sb;
            struct isofs_sb_info    isofs_sb;
            struct nfs_sb_info nfs_sb;
            struct sysv_sb_info     sysv_sb;
            struct affs_sb_info     affs_sb;
            struct ufs_sb_info ufs_sb;
            struct efs_sb_info efs_sb;
            struct shmem_sb_info    shmem_sb;
            struct romfs_sb_info    romfs_sb;
            struct smb_sb_info smbfs_sb;
            struct hfs_sb_info hfs_sb;
```

```
		struct adfs_sb_info	adfs_sb;
		struct qnx4_sb_info	qnx4_sb;
		struct bfs_sb_info	bfs_sb;
		struct udf_sb_info	udf_sb;
		struct ncp_sb_info	ncpfs_sb;
		struct usbdev_sb_info	usbdevfs_sb;
		struct jfs_sb_info	jfs_sb;
		void			*generic_sbp;
	} u;
	/*
	* The next field is for VFS *only*. No filesystems have any business
	 * even looking at it. You had been warned.
	 */
	struct semaphore s_vfs_rename_sem;	/* Kludge */

	/* The next field is used by knfsd when converting a (inode number
	based)
	* file handle into a dentry. As it builds a path in the dcache tree from
	* the bottom up, there may for a time be a subpath of dentrys which is
	not
	* connected to the main tree.  This semaphore ensure that there is only
	ever
	* one such free path per filesystem.  Note that unconnected files (or
	other
	* non-directories) are allowed, but not unconnected directories.
	*/
	struct semaphore s_nfsd_free_path_sem;
};

/*
 * VFS helper functions..
 */

extern int vfs_create(struct inode *,  struct dentry *, int);
extern int vfs_mkdir(struct inode *,   struct dentry *, int);
extern int vfs_mknod(struct inode *,   struct dentry *, int, dev_t);
extern int vfs_symlink(struct inode *, struct dentry *, const char *);
extern int vfs_link(struct dentry *,   struct inode *, struct dentry *);
extern int vfs_rmdir(struct inode *,   struct dentry *);
```

```
extern int vfs_unlink(struct inode *,  struct dentry *);
extern int vfs_rename(struct inode *, struct dentry *, struct inode *, struct
dentry *);

/*
 * File types
 */
#define DT_UNKNOWN	0
#define DT_FIFO		1
#define DT_CHR		2
#define DT_DIR		4
#define DT_BLK		6
#define DT_REG		8
#define DT_LNK		10
#define DT_SOCK		12
#define DT_WHT		14
/*
 * This is the "filldir" function type, used by readdir() to let
 * the kernel specify what kind of direct layout it wants to have.
 * This allows the kernel to read directories into kernel space or
 * to have different direct layouts depending on the binary type.
 */
typedef int (*filldir_t)(void *, const char *, int, off_t, ino_t, unsigned);

struct block_device_operations {
	int (*open) (struct inode *, struct file *);
	int (*release) (struct inode *, struct file *);
	int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
	int (*check_media_change) (kdev_t);
	int (*revalidate) (kdev_t);
};

/*
 * NOTE:
 * read, write, poll, fsync, readv, writev can be called
 *   without the big kernel lock held in all filesystems.
 */
struct file_operations {
	struct module *owner;
```

```
      loff_t (*llseek) (struct file *, loff_t, int);
      ssize_t (*read) (struct file *, char *, size_t, loff_t *);
      ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
      int (*readdir) (struct file *, void *, filldir_t);
      unsigned int (*poll) (struct file *, struct poll_table_struct *);
            int (*ioctl) (struct inode *, struct file *, unsigned int,
      unsigned long);
      int (*mmap) (struct file *, struct vm_area_struct *);
      int (*open) (struct inode *, struct file *);
      int (*flush) (struct file *);
      int (*release) (struct inode *, struct file *);
      int (*fsync) (struct file *, struct dentry *, int datasync);
      int (*fasync) (int, struct file *, int);
      int (*lock) (struct file *, int, struct file_lock *);
      ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
      loff_t *);
      ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
      loff_t *);
};

struct inode_operations {
      int (*create) (struct inode *,struct dentry *,int);
      struct dentry * (*lookup) (struct inode *,struct dentry *);
      int (*link) (struct dentry *,struct inode *,struct dentry *);
      int (*unlink) (struct inode *,struct dentry *);
      int (*symlink) (struct inode *,struct dentry *,const char *);
      int (*mkdir) (struct inode *,struct dentry *,int);
      int (*rmdir) (struct inode *,struct dentry *);
      int (*mknod) (struct inode *,struct dentry *,int,int);
      int (*rename) (struct inode *, struct dentry *,
            struct inode *, struct dentry *);
      int (*readlink) (struct dentry *, char *,int);
      int (*follow_link) (struct dentry *, struct nameidata *);
      void (*truncate) (struct inode *);
      int (*permission) (struct inode *, int);
      int (*revalidate) (struct dentry *);
      int (*setattr) (struct dentry *, struct iattr *);
      int (*getattr) (struct dentry *, struct iattr *);
};
```

```
/*
 * NOTE: write_inode, delete_inode, clear_inode, put_inode can be called
 * without the big kernel lock held in all filesystems.
 */
struct super_operations {
	void (*read_inode) (struct inode *);
	void (*write_inode) (struct inode *, int);
	void (*put_inode) (struct inode *);
	void (*delete_inode) (struct inode *);
	void (*put_super) (struct super_block *);
	void (*write_super) (struct super_block *);
	int (*statfs) (struct super_block *, struct statfs *);
	int (*remount_fs) (struct super_block *, int *, char *);
	void (*clear_inode) (struct inode *);
	void (*umount_begin) (struct super_block *);
};

struct dquot_operations {
	void (*initialize) (struct inode *, short);
	void (*drop) (struct inode *);
	int (*alloc_block) (const struct inode *, unsigned long, char);
	int (*alloc_inode) (const struct inode *, unsigned long);
	void (*free_block) (const struct inode *, unsigned long);
	void (*free_inode) (const struct inode *, unsigned long);
	int (*transfer) (struct dentry *, struct iattr *);
};

struct file_system_type {
	const char *name;
	int fs_flags;
	struct super_block *(*read_super) (struct super_block *, void *, int);
	struct module *owner;
	struct vfsmount *kern_mnt; /* For kernel mount, if it's FS_SINGLE fs */
	struct file_system_type * next;
};

#define DECLARE_FSTYPE(var,type,read,flags) \
struct file_system_type var = { \
```

```
	name:		type, \
	read_super:	read, \
	fs_flags:	flags, \
	owner:			THIS_MODULE, \
}

#define DECLARE_FSTYPE_DEV(var,type,read) \
	DECLARE_FSTYPE(var,type,read,FS_REQUIRES_DEV)

/* Alas, no aliases. Too much hassle with bringing module.h everywhere */
#define fops_get(fops) \
	(((fops) && (fops)->owner)	\
		? ( try_inc_mod_count((fops)->owner) ? (fops) : NULL ) \
		: (fops))

#define fops_put(fops) \
do {	\
	if ((fops) && (fops)->owner) \
		__MOD_DEC_USE_COUNT((fops)->owner); \
} while(0)

extern int register_filesystem(struct file_system_type *);
extern int unregister_filesystem(struct file_system_type *);
extern struct vfsmount *kern_mount(struct file_system_type *);
extern void kern_umount(struct vfsmount *);
extern int may_umount(struct vfsmount *);
extern long do_mount(char *, char *, char *, unsigned long, void *);


extern int vfs_statfs(struct super_block *, struct statfs *);

/* Return value for VFS lock functions - tells locks.c to lock conventionally
 * REALLY kosha for root NFS and nfs_lock
 */
#define LOCK_USE_CLNT 1

#define FLOCK_VERIFY_READ  1
#define FLOCK_VERIFY_WRITE 2
extern int locks_mandatory_locked(struct inode *);
```

```
extern int locks_mandatory_area(int, struct inode *, struct file *, loff_t,
size_t);

/*
 * Candidates for mandatory locking have the setgid bit set
 * but no group execute bit -  an otherwise meaningless combination.
 */
#define MANDATORY_LOCK(inode) \
      (IS_MANDLOCK(inode) && ((inode)->i_mode & (S_ISGID | S_IXGRP)) ==
S_ISGID)

static inline int locks_verify_locked(struct inode *inode)
{
      if (MANDATORY_LOCK(inode))
            return locks_mandatory_locked(inode);
      return 0;
}

static inline int locks_verify_area(int read_write, struct inode *inode,
                            struct file *filp, loff_t offset,
                            size_t count)
{
      if (inode->i_flock && MANDATORY_LOCK(inode))
            return locks_mandatory_area(read_write, inode, filp, offset,
count);
      return 0;
}

static inline int locks_verify_truncate(struct inode *inode,
                            struct file *filp,
                            loff_t size)
{
      if (inode->i_flock && MANDATORY_LOCK(inode))
            return locks_mandatory_area(
                  FLOCK_VERIFY_WRITE, inode, filp,
                  size < inode->i_size ? size : inode->i_size,
                  (size < inode->i_size ? inode->i_size - size
                   : size - inode->i_size)
            );
```

```
	return 0;
}

extern inline int get_lease(struct inode *inode, unsigned int mode)
{
	if (inode->i_flock && (inode->i_flock->fl_flags & FL_LEASE))
		return __get_lease(inode, mode);
	return 0;
}

/* fs/open.c */
asmlinkage long sys_open(const char *, int, int);
asmlinkage long sys_close(unsigned int);   /* yes, it's really unsigned */
extern int do_truncate(struct dentry *, loff_t start);

extern struct file *filp_open(const char *, int, int);
extern struct file * dentry_open(struct dentry *, struct vfsmount *, int);
extern int filp_close(struct file *, fl_owner_t id);
extern char * getname(const char *);

/* fs/dcache.c */
extern void vfs_caches_init(unsigned long);

#define __getname()      kmem_cache_alloc(names_cachep, SLAB_KERNEL)
#define putname(name)    kmem_cache_free(names_cachep, (void *)(name))

enum {BDEV_FILE, BDEV_SWAP, BDEV_FS, BDEV_RAW};
extern int register_blkdev(unsigned int, const char *, struct
block_device_operations *);
extern int unregister_blkdev(unsigned int, const char *);
extern struct block_device *bdget(dev_t);
extern void bdput(struct block_device *);
extern int blkdev_open(struct inode *, struct file *);
extern struct file_operations def_blk_fops;
extern struct file_operations def_fifo_fops;
extern int ioctl_by_bdev(struct block_device *, unsigned, unsigned long);
extern int blkdev_get(struct block_device *, mode_t, unsigned, int);
extern int blkdev_put(struct block_device *, int);
```

```
/* fs/devices.c */
extern const struct block_device_operations *get_blkfops(unsigned int);
extern int register_chrdev(unsigned int, const char *, struct
file_operations *);
extern int unregister_chrdev(unsigned int, const char *);
extern int chrdev_open(struct inode *, struct file *);
extern const char * bdevname(kdev_t);
extern const char * cdevname(kdev_t);
extern const char * kdevname(kdev_t);
extern void init_special_inode(struct inode *, umode_t, int);

/* Invalid inode operations -- fs/bad_inode.c */
extern void make_bad_inode(struct inode *);
extern int is_bad_inode(struct inode *);

extern struct file_operations read_fifo_fops;
extern struct file_operations write_fifo_fops;
extern struct file_operations rdwr_fifo_fops;
extern struct file_operations read_pipe_fops;
extern struct file_operations write_pipe_fops;
extern struct file_operations rdwr_pipe_fops;

extern int fs_may_remount_ro(struct super_block *);

extern int try_to_free_buffers(struct page *, int);
extern void refile_buffer(struct buffer_head * buf);
#define BUF_CLEAN 0
#define BUF_LOCKED    1 /* Buffers scheduled for write */
#define BUF_DIRTY     2 /* Dirty buffers, not yet scheduled for write */
#define BUF_PROTECTED 3 /* Ramdisk persistent storage */
#define NR_LIST       4

/*
 * This is called by bh->b_end_io() handlers when I/O has completed.
 */
static inline void mark_buffer_uptodate(struct buffer_head * bh, int on)
{
	if (on)
		set_bit(BH_Uptodate, &bh->b_state);
```

```
	else
		clear_bit(BH_Uptodate, &bh->b_state);
}

#define atomic_set_buffer_clean(bh) test_and_clear_bit(BH_Dirty, &(bh)-
>b_state)

static inline void __mark_buffer_clean(struct buffer_head *bh)
{
	refile_buffer(bh);
}
static inline void mark_buffer_clean(struct buffer_head * bh)
{
	if (atomic_set_buffer_clean(bh))
		__mark_buffer_clean(bh);
}

#define atomic_set_buffer_protected(bh) test_and_set_bit(BH_Protected,
&(bh)->b_state)

static inline void __mark_buffer_protected(struct buffer_head *bh)
{
	refile_buffer(bh);
}

static inline void mark_buffer_protected(struct buffer_head * bh)
{
	if (!atomic_set_buffer_protected(bh))
		__mark_buffer_protected(bh);
}

extern void FASTCALL(__mark_buffer_dirty(struct buffer_head *bh));
extern void FASTCALL(mark_buffer_dirty(struct buffer_head *bh));

#define atomic_set_buffer_dirty(bh) test_and_set_bit(BH_Dirty, &(bh)-
>b_state)

/*
 * If an error happens during the make_request, this function
```

```
 * has to be recalled. It marks the buffer as clean and not
 * uptodate, and it notifys the upper layer about the end
 * of the I/O.
 */
static inline void buffer_IO_error(struct buffer_head * bh)
{
	mark_buffer_clean(bh);
	/*
	 * b_end_io has to clear the BH_Uptodate bitflag in the error case!
	 */
	bh->b_end_io(bh, 0);
}

extern void buffer_insert_inode_queue(struct buffer_head *, struct inode *);
static inline void mark_buffer_dirty_inode(struct buffer_head *bh, struct
inode *inode)
{
	mark_buffer_dirty(bh);
	buffer_insert_inode_queue(bh, inode);
}

extern void balance_dirty(kdev_t);
extern int check_disk_change(kdev_t);
extern int invalidate_inodes(struct super_block *);
extern void invalidate_inode_pages(struct inode *);
extern void invalidate_inode_buffers(struct inode *);
#define invalidate_buffers(dev)	__invalidate_buffers((dev), 0)
#define destroy_buffers(dev)	__invalidate_buffers((dev), 1)
extern void __invalidate_buffers(kdev_t dev, int);
extern void sync_inodes(kdev_t);
extern void write_inode_now(struct inode *, int);
extern void sync_dev(kdev_t);
extern int fsync_dev(kdev_t);
extern int fsync_inode_buffers(struct inode *);
extern int osync_inode_buffers(struct inode *);
extern int inode_has_buffers(struct inode *);
extern void filemap_fdatasync(struct address_space *);
extern void filemap_fdatawait(struct address_space *);
```

```
extern void sync_supers(kdev_t);
extern int bmap(struct inode *, int);
extern int notify_change(struct dentry *, struct iattr *);
extern int permission(struct inode *, int);
extern int vfs_permission(struct inode *, int);
extern int get_write_access(struct inode *);
extern int deny_write_access(struct file *);
static inline void put_write_access(struct inode * inode)
{
	atomic_dec(&inode->i_writecount);
}
static inline void allow_write_access(struct file *file)
{
	if (file)
		atomic_inc(&file->f_dentry->d_inode->i_writecount);
}
extern int do_pipe(int *);

extern int open_namei(const char *, int, int, struct nameidata *);

extern int kernel_read(struct file *, unsigned long, char *, unsigned long);
extern struct file * open_exec(const char *);

/* fs/dcache.c -- generic fs support functions */
extern int is_subdir(struct dentry *, struct dentry *);
extern ino_t find_inode_number(struct dentry *, struct qstr *);

/*
 * Kernel pointers have redundant information, so we can use a
 * scheme where we can return either an error code or a dentry
 * pointer with the same return value.
 *
 * This should be a per-architecture thing, to allow different
 * error and pointer decisions.
 */
static inline void *ERR_PTR(long error)
{
	return (void *) error;
```

```
}

static inline long PTR_ERR(const void *ptr)
{
      return (long) ptr;
}

static inline long IS_ERR(const void *ptr)
{
      return (unsigned long)ptr > (unsigned long)-1000L;
}

/*
 * The bitmask for a lookup event:
 *  - follow links at the end
 *  - require a directory
 *  - ending slashes ok even for nonexistent files
 *  - internal "there are more path components" flag
 */
#define LOOKUP_FOLLOW          (1)
#define LOOKUP_DIRECTORY       (2)
#define LOOKUP_CONTINUE        (4)
#define LOOKUP_POSITIVE        (8)
#define LOOKUP_PARENT          (16)
#define LOOKUP_NOALT           (32)
/*
 * Type of the last component on LOOKUP_PARENT
 */
enum {LAST_NORM, LAST_ROOT, LAST_DOT, LAST_DOTDOT, LAST_BIND};

/*
 * "descriptor" for what we're up to with a read for sendfile().
 * This allows us to use the same read code yet
 * have multiple different users of the data that
 * we read from a file.
 *
 * The simplest case just copies the data to user
 * mode.
 */
```

```
typedef struct {
      size_t written;
      size_t count;
      char * buf;
      int error;
} read_descriptor_t;

typedef int (*read_actor_t)(read_descriptor_t *, struct page *, unsigned
long, unsigned long);

/* needed for stackable file system support */
extern loff_t default_llseek(struct file *file, loff_t offset, int origin);

extern int __user_walk(const char *, unsigned, struct nameidata *);
extern int path_init(const char *, unsigned, struct nameidata *);
extern int path_walk(const char *, struct nameidata *);
extern void path_release(struct nameidata *);
extern int follow_down(struct vfsmount **, struct dentry **);
extern int follow_up(struct vfsmount **, struct dentry **);
extern struct dentry * lookup_one(const char *, struct dentry *);
extern struct dentry * lookup_hash(struct qstr *, struct dentry *);
#define user_path_walk(name,nd) __user_walk(name,
LOOKUP_FOLLOW|LOOKUP_POSITIVE, nd)
#define user_path_walk_link(name,nd) __user_walk(name, LOOKUP_POSITIVE, nd)

extern void iput(struct inode *);
extern void force_delete(struct inode *);
extern struct inode * igrab(struct inode *);
extern ino_t iunique(struct super_block *, ino_t);

typedef int (*find_inode_t)(struct inode *, unsigned long, void *);
extern struct inode * iget4(struct super_block *, unsigned long, find_inode_t,
void *);
static inline struct inode *iget(struct super_block *sb, unsigned long ino)
{
      return iget4(sb, ino, NULL, NULL);
}
extern void clear_inode(struct inode *);
extern struct inode * get_empty_inode(void);
```

```
static inline struct inode * new_inode(struct super_block *sb)
{
	struct inode *inode = get_empty_inode();
	if (inode) {
		inode->i_sb = sb;
		inode->i_dev = sb->s_dev;
	}
	return inode;
}

extern void insert_inode_hash(struct inode *);
extern void remove_inode_hash(struct inode *);
extern struct file * get_empty_filp(void);
extern void file_move(struct file *f, struct list_head *list);
extern void file_moveto(struct file *new, struct file *old);
extern struct buffer_head * get_hash_table(kdev_t, int, int);
extern struct buffer_head * getblk(kdev_t, int, int);
extern void ll_rw_block(int, int, struct buffer_head * bh[]);
extern void submit_bh(int, struct buffer_head *);
extern int is_read_only(kdev_t);
extern void __brelse(struct buffer_head *);
static inline void brelse(struct buffer_head *buf)
{
	if (buf)
		__brelse(buf);
}
extern void __bforget(struct buffer_head *);
static inline void bforget(struct buffer_head *buf)
{
	if (buf)
		__bforget(buf);
}
extern void set_blocksize(kdev_t, int);
extern unsigned int get_hardblocksize(kdev_t);
extern struct buffer_head * bread(kdev_t, int, int);
extern void wakeup_bdflush(int wait);

extern int brw_page(int, struct page *, kdev_t, int [], int);
```

```
typedef int (get_block_t)(struct inode*,long,struct buffer_head*,int);

/* Generic buffer handling for block filesystems.. */
extern int block_flushpage(struct page *, unsigned long);
extern int block_symlink(struct inode *, const char *, int);
extern int block_write_full_page(struct page*, get_block_t*);
extern int block_read_full_page(struct page*, get_block_t*);
extern int block_prepare_write(struct page*, unsigned, unsigned,
get_block_t*);
extern int cont_prepare_write(struct page*, unsigned, unsigned, get_block_t*,
                        unsigned long *);
extern int block_sync_page(struct page *);

int generic_block_bmap(struct address_space *, long, get_block_t *);
int generic_commit_write(struct file *, struct page *, unsigned, unsigned);
int block_truncate_page(struct address_space *, loff_t, get_block_t *);

extern int generic_file_mmap(struct file *, struct vm_area_struct *);
extern ssize_t generic_file_read(struct file *, char *, size_t, loff_t *);
extern ssize_t generic_file_write(struct file *, const char *, size_t, loff_t
*);
extern void do_generic_file_read(struct file *, loff_t *, read_descriptor_t
*, read_actor_t);

extern ssize_t generic_read_dir(struct file *, char *, size_t, loff_t *);

extern struct file_operations generic_ro_fops;

extern int vfs_readlink(struct dentry *, char *, int, const char *);
extern int vfs_follow_link(struct nameidata *, const char *);
extern int page_readlink(struct dentry *, char *, int);
extern int page_follow_link(struct dentry *, struct nameidata *);
extern struct inode_operations page_symlink_inode_operations;

extern int vfs_readdir(struct file *, filldir_t, void *);
extern int dcache_readdir(struct file *, void *, filldir_t);

extern struct file_system_type *get_fs_type(const char *name);
extern struct super_block *get_super(kdev_t);
```

```
struct super_block *get_empty_super(void);
extern void put_super(kdev_t);
unsigned long generate_cluster(kdev_t, int b[], int);
unsigned long generate_cluster_swab32(kdev_t, int b[], int);
extern kdev_t ROOT_DEV;
extern char root_device_name[];

extern void show_buffers(void);
extern void mount_root(void);
#ifdef CONFIG_BLK_DEV_INITRD
extern kdev_t real_root_dev;
extern int change_root(kdev_t, const char *);
#endif

extern ssize_t char_read(struct file *, char *, size_t, loff_t *);
extern ssize_t block_read(struct file *, char *, size_t, loff_t *);
extern int read_ahead[];

extern ssize_t char_write(struct file *, const char *, size_t, loff_t *);
extern ssize_t block_write(struct file *, const char *, size_t, loff_t *);

extern int file_fsync(struct file *, struct dentry *, int);
extern int generic_buffer_fdatasync(struct inode *inode, unsigned long
start_idx, unsigned long end_idx);
extern int generic_osync_inode(struct inode *, int);

extern int inode_change_ok(struct inode *, struct iattr *);
extern void inode_setattr(struct inode *, struct iattr *);

/*
 * Common dentry functions for inclusion in the VFS
 * or in other stackable file systems.  Some of these
 * functions were in linux/fs/ C (VFS) files.
 *
 */

/*
 * Locking the parent is needed to:
 *  - serialize directory operations
```

```
 *  - make sure the parent doesn't change from
 *    under us in the middle of an operation.
 *
 * NOTE! Right now we'd rather use a "struct inode"
 * for this, but as I expect things to move toward
 * using dentries instead for most things it is
 * probably better to start with the conceptually
 * better interface of relying on a path of dentries.
 */
static inline struct dentry *lock_parent(struct dentry *dentry)
{
	struct dentry *dir = dget(dentry->d_parent);

	down(&dir->d_inode->i_sem);
	return dir;
}

static inline struct dentry *get_parent(struct dentry *dentry)
{
	return dget(dentry->d_parent);
}

static inline void unlock_dir(struct dentry *dir)
{
	up(&dir->d_inode->i_sem);
	dput(dir);
}

/*
 * Whee.. Deadlock country. Happily there are only two VFS
 * operations that does this..
 */
static inline void double_down(struct semaphore *s1, struct semaphore *s2)
{
	if (s1 != s2) {
		if ((unsigned long) s1 < (unsigned long) s2) {
			struct semaphore *tmp = s2;
			s2 = s1; s1 = tmp;
		}
```

```
		down(s1);
	}
	down(s2);
}

/*
 * Ewwwwwwww... _triple_ lock. We are guaranteed that the 3rd argument is
 * not equal to 1st and not equal to 2nd - the first case (target is parent of
 * source) would be already caught, the second is plain impossible (target is
 * its own parent and that case would be caught even earlier). Very messy.
 * I _think_ that it works, but no warranties - please, look it through.
 * Pox on bloody lusers who mandated overwriting rename() for directories...
 */

static inline void triple_down(struct semaphore *s1,
			       struct semaphore *s2,
			       struct semaphore *s3)
{

	if (s1 != s2) {
		if ((unsigned long) s1 < (unsigned long) s2) {
			if ((unsigned long) s1 < (unsigned long) s3) {
				struct semaphore *tmp = s3;
				s3 = s1; s1 = tmp;
			}
			if ((unsigned long) s1 < (unsigned long) s2) {
				struct semaphore *tmp = s2;
				s2 = s1; s1 = tmp;
			}
		} else {
			if ((unsigned long) s1 < (unsigned long) s3) {
				struct semaphore *tmp = s3;
				s3 = s1; s1 = tmp;
			}
			if ((unsigned long) s2 < (unsigned long) s3) {
				struct semaphore *tmp = s3;
				s3 = s2; s2 = tmp;
			}
		}
```

```
            down(s1);
      } else if ((unsigned long) s2 < (unsigned long) s3) {
            struct semaphore *tmp = s3;
            s3 = s2; s2 = tmp;
      }
      down(s2);
      down(s3);
}

static inline void double_up(struct semaphore *s1, struct semaphore *s2)
{
      up(s1);
      if (s1 != s2)
            up(s2);
}

static inline void triple_up(struct semaphore *s1,
                      struct semaphore *s2,
                      struct semaphore *s3)
{
      up(s1);
      if (s1 != s2)
            up(s2);
      up(s3);
}

static inline void double_lock(struct dentry *d1, struct dentry *d2)
{
      double_down(&d1->d_inode->i_sem, &d2->d_inode->i_sem);
}

static inline void double_unlock(struct dentry *d1, struct dentry *d2)
{
      double_up(&d1->d_inode->i_sem,&d2->d_inode->i_sem);
      dput(d1);
      dput(d2);
}

#endif /* __KERNEL__ */
```

```
#endif /* _Linux_FS_H */
```

원천파일 fs/ext2/super.c(2.4.3)

```
/*
 *  linux/fs/ext2/super.c
 *
 * Copyright (C) 1992, 1993, 1994, 1995
 * Remy Card (card@masi.ibp.fr)
 * Laboratoire MASI - Institut Blaise Pascal
 * Universite Pierre et Marie Curie (Paris VI)
 *
 *  from
 *
 *  linux/fs/minix/inode.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  Big-endian to little-endian byte-swapping/bitmaps by
 *        David S. Miller (davem@caip.rutgers.edu), 1995
 */

#include <linux/config.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/ext2_fs.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/locks.h>
#include <asm/uaccess.h>

static char error_buf[1024];

void ext2_error (struct super_block * sb, const char * function,
             const char * fmt, ...)
{
      va_list args;
```

```
	if (!(sb->s_flags & MS_RDONLY)) {
		sb->u.ext2_sb.s_mount_state |= EXT2_ERROR_FS;
		sb->u.ext2_sb.s_es->s_state =
			cpu_to_le16(le16_to_cpu(sb->u.ext2_sb.s_es->s_state) |
EXT2_ERROR_FS);
		mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
		sb->s_dirt = 1;
	}
	va_start (args, fmt);
	vsprintf (error_buf, fmt, args);
	va_end (args);
	if (test_opt (sb, ERRORS_PANIC) ||
	    (le16_to_cpu(sb->u.ext2_sb.s_es->s_errors) == EXT2_ERRORS_PANIC
&&
	     !test_opt (sb, ERRORS_CONT) && !test_opt (sb, ERRORS_RO)))
		panic ("EXT2-fs panic (device %s): %s: %s\n",
		       bdevname(sb->s_dev), function, error_buf);
	printk (KERN_CRIT "EXT2-fs error (device %s): %s: %s\n",
		bdevname(sb->s_dev), function, error_buf);
	if (test_opt (sb, ERRORS_RO) ||
	    (le16_to_cpu(sb->u.ext2_sb.s_es->s_errors) == EXT2_ERRORS_RO &&
	     !test_opt (sb, ERRORS_CONT) && !test_opt (sb, ERRORS_PANIC))) {
		printk ("Remounting filesystem read-only\n");
		sb->s_flags |= MS_RDONLY;
	}
}

NORET_TYPE void ext2_panic (struct super_block * sb, const char *
function,
			    const char * fmt, ...)
{
	va_list args;

	if (!(sb->s_flags & MS_RDONLY)) {
		sb->u.ext2_sb.s_mount_state |= EXT2_ERROR_FS;
		sb->u.ext2_sb.s_es->s_state =
			cpu_to_le16(le16_to_cpu(sb->u.ext2_sb.s_es->s_state) |
EXT2_ERROR_FS);
```

```
		mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
		sb->s_dirt = 1;
	}
	va_start (args, fmt);
	vsprintf (error_buf, fmt, args);
	va_end (args);
	/* this is to prevent panic from syncing this filesystem */
	if (sb->s_lock)
		sb->s_lock=0;
	sb->s_flags |= MS_RDONLY;
	panic ("EXT2-fs panic (device %s): %s: %s\n",
		bdevname(sb->s_dev), function, error_buf);
}

void ext2_warning (struct super_block * sb, const char * function,
		   const char * fmt, ...)
{
	va_list args;

	va_start (args, fmt);
	vsprintf (error_buf, fmt, args);
	va_end (args);
	printk (KERN_WARNING "EXT2-fs warning (device %s): %s: %s\n",
		bdevname(sb->s_dev), function, error_buf);
}
void ext2_update_dynamic_rev(struct super_block *sb)
{
	struct ext2_super_block *es = EXT2_SB(sb)->s_es;

	if (le32_to_cpu(es->s_rev_level) > EXT2_GOOD_OLD_REV)
		return;

	ext2_warning(sb, __FUNCTION__,
		     "updating to rev %d because of new feature flag, "
		     "running e2fsck is recommended",
		     EXT2_DYNAMIC_REV);

	es->s_first_ino = cpu_to_le32(EXT2_GOOD_OLD_FIRST_INO);
	es->s_inode_size = cpu_to_le16(EXT2_GOOD_OLD_INODE_SIZE);
```

```
        es->s_rev_level = cpu_to_le32(EXT2_DYNAMIC_REV);
        /* leave es->s_feature_*compat flags alone */
        /* es->s_uuid will be set by e2fsck if empty */

        /*
         * The rest of the superblock fields should be zero, and if not it
         * means they are likely already in use, so leave them alone.  We
         * can leave it up to e2fsck to clean up any inconsistencies there.
         */
}

void ext2_put_super (struct super_block * sb)
{
        int db_count;
        int i;

        if (!(sb->s_flags & MS_RDONLY)) {
                sb->u.ext2_sb.s_es->s_state = le16_to_cpu(sb-
>u.ext2_sb.s_mount_state);
                mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
        }
        db_count = EXT2_SB(sb)->s_gdb_count;
        for (i = 0; i < db_count; i++)
                if (sb->u.ext2_sb.s_group_desc[i])
                        brelse (sb->u.ext2_sb.s_group_desc[i]);
        kfree(sb->u.ext2_sb.s_group_desc);
        for (i = 0; i < EXT2_MAX_GROUP_LOADED; i++)
                if (sb->u.ext2_sb.s_inode_bitmap[i])
                        brelse (sb->u.ext2_sb.s_inode_bitmap[i]);
        for (i = 0; i < EXT2_MAX_GROUP_LOADED; i++)
                if (sb->u.ext2_sb.s_block_bitmap[i])
                        brelse (sb->u.ext2_sb.s_block_bitmap[i]);
        brelse (sb->u.ext2_sb.s_sbh);

        return;
}

static struct super_operations ext2_sops = {
        read_inode:     ext2_read_inode,
```

```
	write_inode:	ext2_write_inode,
	put_inode:	ext2_put_inode,
	delete_inode:	ext2_delete_inode,
	put_super:	ext2_put_super,
	write_super:	ext2_write_super,
	statfs:		ext2_statfs,
	remount_fs:	ext2_remount,
};

/*
 * This function has been shamelessly adapted from the msdos fs
 */
static int parse_options (char * options, unsigned long * b_block,
			  unsigned short *resuid, unsigned short * resgid,
			  unsigned long * mount_options)
{
	char * this_char;
	char * value;

	if (!options)
		return 1;
	for (this_char = strtok (options, ",");
	     this_char != NULL;
	     this_char = strtok (NULL, ",")) {
		if ((value = strchr (this_char, '=')) != NULL)
			*value++ = 0;
		if (!strcmp (this_char, "bsddf"))
			clear_opt (*mount_options, MINIX_DF);
		else if (!strcmp (this_char, "nouid32")) {
			set_opt (*mount_options, NO_UID32);
		}
		else if (!strcmp (this_char, "check")) {
			if (!value || !*value || !strcmp (value, "none"))
				clear_opt (*mount_options, CHECK);
			else
#ifdef CONFIG_EXT2_CHECK
				set_opt (*mount_options, CHECK);
#else
			printk("EXT2 Check option not supported\n");
```

```
#endif
        }
        else if (!strcmp (this_char, "debug"))
            set_opt (*mount_options, DEBUG);
        else if (!strcmp (this_char, "errors")) {
            if (!value || !*value) {
            printk ("EXT2-fs: the errors option requires "
                    "an argument\n");
                return 0;
            }
            if (!strcmp (value, "continue")) {
                clear_opt (*mount_options, ERRORS_RO);
                clear_opt (*mount_options, ERRORS_PANIC);
                set_opt (*mount_options, ERRORS_CONT);
            }
            else if (!strcmp (value, "remount-ro")) {
                clear_opt (*mount_options, ERRORS_CONT);
                clear_opt (*mount_options, ERRORS_PANIC);
                set_opt (*mount_options, ERRORS_RO);
            }
            else if (!strcmp (value, "panic")) {
                clear_opt (*mount_options, ERRORS_CONT);
                clear_opt (*mount_options, ERRORS_RO);
                set_opt (*mount_options, ERRORS_PANIC);
            }
            else {
            printk ("EXT2-fs: Invalid errors option: %s\n",
                    value);
                return 0;
            }
        }
        else if (!strcmp (this_char, "grpid") ||
             !strcmp (this_char, "bsdgroups"))
            set_opt (*mount_options, GRPID);
        else if (!strcmp (this_char, "minixdf"))
            set_opt (*mount_options, MINIX_DF);
        else if (!strcmp (this_char, "nocheck"))
            clear_opt (*mount_options, CHECK);
        else if (!strcmp (this_char, "nogrpid") ||
```

```
             !strcmp (this_char, "sysvgroups"))
            clear_opt (*mount_options, GRPID);
      else if (!strcmp (this_char, "resgid")) {
            if (!value || !*value) {
            printk ("EXT2-fs: the resgid option requires "
                        "an argument\n");
                  return 0;
            }
            *resgid = simple_strtoul (value, &value, 0);
            if (*value) {
            printk ("EXT2-fs: Invalid resgid option: %s\n",
                        value);
                  return 0;
            }
      }
      else if (!strcmp (this_char, "resuid")) {
            if (!value || !*value) {
            printk ("EXT2-fs: the resuid option requires "
                        "an argument");
                  return 0;
            }
            *resuid = simple_strtoul (value, &value, 0);
            if (*value) {
            printk ("EXT2-fs: Invalid resuid option: %s\n",
                        value);
                  return 0;
            }
      }
      else if (!strcmp (this_char, "sb")) {
            if (!value || !*value) {
                  printk ("EXT2-fs: the sb option requires "
                        "an argument");
                  return 0;
            }
            *sb_block = simple_strtoul (value, &value, 0);
            if (*value) {
                  printk ("EXT2-fs: Invalid sb option: %s\n",
                        value);
                  return 0;
```

```
            }
        }
        /* Silently ignore the quota options */
        else if (!strcmp (this_char, "grpquota")
                 || !strcmp (this_char, "noquota")
                 || !strcmp (this_char, "quota")
                 || !strcmp (this_char, "usrquota"))
            /* Don't do anything ;-) */ ;
        else {
    printk ("EXT2-fs: Unrecognized mount option %s\n", this_char);
            return 0;
        }
    }
    return 1;
}

static int ext2_setup_super (struct super_block * sb,
                             struct ext2_super_block * es,
                             int read_only)
{
    int res = 0;
    if (le32_to_cpu(es->s_rev_level) > EXT2_MAX_SUPP_REV) {
        printk ("EXT2-fs warning: revision level too high, "
            "forcing read-only mode\n");
        res = MS_RDONLY;
    }
    if (read_only)
        return res;
    if (!(sb->u.ext2_sb.s_mount_state & EXT2_VALID_FS))
        printk ("EXT2-fs warning: mounting unchecked fs, "
            "running e2fsck is recommended\n");
    else if ((sb->u.ext2_sb.s_mount_state & EXT2_ERROR_FS))
        printk ("EXT2-fs warning: mounting fs with errors, "
            "running e2fsck is recommended\n");
    else if ((__s16) le16_to_cpu(es->s_max_mnt_count) >= 0 &&
         le16_to_cpu(es->s_mnt_count) >=
    (unsigned short) (__s16) le16_to_cpu(es->s_max_mnt_count))
    printk ("EXT2-fs warning: maximal mount count reached, "
            "running e2fsck is recommended\n");
```

```
	else if (le32_to_cpu(es->s_checkinterval) &&
			(le32_to_cpu(es->s_lastcheck) + le32_to_cpu(es-
	   >s_checkinterval) <= CURRENT_TIME))
		printk ("EXT2-fs warning: checktime reached, "
			"running e2fsck is recommended\n");
	es->s_state = cpu_to_le16(le16_to_cpu(es->s_state) & ~EXT2
		_VALID_FS);
	if (!(__s16) le16_to_cpu(es->s_max_mnt_count))
		es->s_max_mnt_count = (__s16) cpu_to_le16(EXT2_DFL_MAX
	_MNT_COUNT);
	es->s_mnt_count=cpu_to_le16(le16_to_cpu(es->s_mnt_count)+);
	es->s_mtime = cpu_to_le32(CURRENT_TIME);
	mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
	sb->s_dirt = 1;
	if (test_opt (sb, DEBUG))
		printk ("[EXT II FS %s, %s, bs=%lu, fs=%lu, gc=%lu, "
			"bpg=%lu, ipg=%lu, mo=%04lx]\n",
			EXT2FS_VERSION, EXT2FS_DATE, sb->s_blocksize,
			sb->u.ext2_sb.s_frag_size,
			sb->u.ext2_sb.s_groups_count,
			EXT2_BLOCKS_PER_GROUP(sb),
			EXT2_INODES_PER_GROUP(sb),
			sb->u.ext2_sb.s_mount_opt);
#ifdef CONFIG_EXT2_CHECK
	if (test_opt (sb, CHECK)) {
		ext2_check_blocks_bitmap (sb);
		ext2_check_inodes_bitmap (sb);
	}
#endif
	return res;
}

static int ext2_check_descriptors (struct super_block * sb)
{
	int i;
	int desc_block = 0;
	unsigned long block = le32_to_cpu(sb->u.ext2_sb.s_es-
	>s_first_data_block);
	struct ext2_group_desc * gdp = NULL;
```

```
ext2_debug ("Checking group descriptors");

for (i = 0; i < sb->u.ext2_sb.s_groups_count; i++)
{
      if ((i % EXT2_DESC_PER_BLOCK(sb)) == 0)
         gdp = (struct ext2_group_desc *) sb-
         >u.ext2_sb.s_group_desc[desc_block++]->b_data;
      if (le32_to_cpu(gdp->bg_block_bitmap) < block ||
        le32_to_cpu(gdp->bg_block_bitmap) >= block +
        EXT2_BLOCKS_PER_GROUP(sb))
                  {
                        ext2_error (sb, "ext2_check_descriptors",
                      "Block bitmap for group %d"
                      " not in group (block %lu)!",
                        i,(unsigned long) le32_to_cpu(gdp-
           >bg_block_bitmap));
            return 0;
      }
      if (le32_to_cpu(gdp->bg_inode_bitmap) < block ||
           le32_to_cpu(gdp->bg_inode_bitmap) >= block +
           EXT2_BLOCKS_PER_GROUP(sb))
      {
            ext2_error (sb, "ext2_check_descriptors",
                      "Inode bitmap for group %d"
                      " not in group (block %lu)!",
                     i, (unsigned long) le32_to_cpu(gdp-
                     >bg_inode_bitmap));
            return 0;
      }
      if (le32_to_cpu(gdp->bg_inode_table) < block ||
            le32_to_cpu(gdp->bg_inode_table) + sb-
           >u.ext2_sb.s_itb_per_group >=
         block + EXT2_BLOCKS_PER_GROUP(sb))
      {
            ext2_error (sb, "ext2_check_descriptors",
                      "Inode table for group %d"
                      " not in group (block %lu)!",
                        i, (unsigned long) le32_to_cpu(gdp-
```

```
                            >bg_inode_table));
                    return 0;
              }
              block += EXT2_BLOCKS_PER_GROUP(sb);
              gdp++;
       }
       return 1;
}

#define log2(n) ffz(~(n))

/*
 * Maximal file size.  There is a direct, and {,double-,triple-}indirect
              * block limit, and also a limit of (2^32 - 1) 512-byte
              sectors in i_blocks.
 * We need to be 1 filesystem block less than the 2^32 sector limit.
 */
static loff_t ext2_max_size(int bits)
{
       loff_t res = EXT2_NDIR_BLOCKS;
       res += 1LL << (bits-2);
       res += 1LL << (2*(bits-2));
       res += 1LL << (3*(bits-2));
       res <<= bits;
       if (res > (512LL << 32) - (1 << bits))
       res = (512LL << 32) - (1 << bits);
       return res;
}

struct super_block * ext2_read_super (struct super_block * sb, void *
data,int silent)
{
       struct buffer_head * bh;
       struct ext2_super_block * es;
       unsigned long sb_block = 1;
       unsigned short resuid = EXT2_DEF_RESUID;
       unsigned short resgid = EXT2_DEF_RESGID;
       unsigned long logic_sb_block = 1;
       unsigned long offset = 0;
```

```
kdev_t dev = sb->s_dev;
int blocksize;
int hblock;
int db_count;
int i, j;

/*
 * See what the current blocksize for the device is, and
 * use that as the blocksize.  Otherwise (or if the blocksize
 * is smaller than the default) use the default.
 * This is important for devices that have a hardware
 * sectorsize that is larger than the default.
 */
blocksize = BLOCK_SIZE;
hblock = get_hardblocksize(dev);
if (blocksize < hblock)
      blocksize = hblock;

sb->u.ext2_sb.s_mount_opt = 0;
if (!parse_options ((char *) data, &sb_block, &resuid, &resgid,
    &sb->u.ext2_sb.s_mount_opt)) {
      return NULL;
}

set_blocksize (dev, blocksize);

/*
 * If the superblock doesn't start on a sector boundary,
 * calculate the offset.  FIXME(eric) this doesn't make sense
 * that we would have to do this.
 */
if (blocksize != BLOCK_SIZE) {
      logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
      offset = (sb_block*BLOCK_SIZE) % blocksize;
}

if (!(bh = bread (dev, logic_sb_block, blocksize))) {
      printk ("EXT2-fs: unable to read superblock\n");
      return NULL;
```

```
}
/*
* Note: s_es must be initialized s_es as soon as possible because
* some ext2 macro-instructions depend on its value
*/
es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
sb->u.ext2_sb.s_es = es;
sb->s_magic = le16_to_cpu(es->s_magic);
if (sb->s_magic != EXT2_SUPER_MAGIC) {
      if (!silent)
      printk ("VFS: Can't find an ext2 filesystem on dev "
                  "%s.\n", bdevname(dev));
failed_mount:
      if (bh)
            brelse(bh);
      return NULL;
}
if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV &&
    (EXT2_HAS_COMPAT_FEATURE(sb, ~0U) ||
     EXT2_HAS_RO_COMPAT_FEATURE(sb, ~0U) ||
     EXT2_HAS_INCOMPAT_FEATURE(sb, ~0U)))
printk("EXT2-fs warning: feature flags set on rev 0 fs, "
            "running e2fsck is recommended\n");
/*
 * Check feature flags regardless of the revision level,
   since we
   * previously didn't change the revision level when setting the
   flags,
  * so there is a chance incompat flags are set on a rev 0
  filesystem.
 */
if ((i=EXT2_HAS_INCOMPAT_FEATURE(sb, ~EXT2_FEATURE_INCOMPAT_SUPP))){
      printk("EXT2-fs: %s: couldn't mount because of "
            "unsupported optional features (%x).\n",
            bdevname(dev), i);
      goto failed_mount;
}
if (!(sb->s_flags & MS_RDONLY) &&(i=EXT2_HAS_RO_COMPAT_FEATURE
      (sb, ~EXT2_FEATURE_RO_COMPAT_SUPP))){
```

```
		printk("EXT2-fs: %s: couldn't mount RDWR because of "
		       "unsupported optional features (%x).\n",
		       bdevname(dev), i);
		goto failed_mount;
	}
	sb->s_blocksize_bits =
		le32_to_cpu(EXT2_SB(sb)->s_es->s_log_block_size) + 10;
	sb->s_blocksize = 1 << sb->s_blocksize_bits;
	sb->s_maxbytes = ext2_max_size(sb->s_blocksize_bits);

	if (sb->s_blocksize != blocksize &&
	    (sb->s_blocksize == 1024 || sb->s_blocksize == 2048 ||
	     sb->s_blocksize == 4096)) {
		/*
		 * Make sure the blocksize for the filesystem is larger
		 * than the hardware sectorsize for the machine.
		 */
		if (sb->s_blocksize < hblock) {
		printk("EXT2-fs: blocksize too small for device.\n");
			goto failed_mount;
		}

		brelse (bh);
		set_blocksize (dev, sb->s_blocksize);
	logic_sb_block = (sb_block*BLOCK_SIZE) / sb->s_blocksize;
		offset = (sb_block*BLOCK_SIZE) % sb->s_blocksize;
		bh = bread (dev, logic_sb_block, sb->s_blocksize);
		if(!bh) {
			printk("EXT2-fs: Couldn't read superblock on "
			       "2nd try.\n");
			goto failed_mount;
		}
es=(struct ext2_super_block *) (((char *)bh->b_data)+offset);
		sb->u.ext2_sb.s_es = es;
		if (es->s_magic != le16_to_cpu(EXT2_SUPER_MAGIC)) {
		printk ("EXT2-fs: Magic mismatch, very weird !\n");
			goto failed_mount;
		}
	}
```

```
	if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV) {
		sb->u.ext2_sb.s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
		sb->u.ext2_sb.s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
	} else {
	sb->u.ext2_sb.s_inode_size = le16_to_cpu(es->s_inode_size);
	sb->u.ext2_sb.s_first_ino = le32_to_cpu(es->s_first_ino);
	if (sb->u.ext2_sb.s_inode_size != EXT2_GOOD_OLD_INODE_SIZE)
{
			printk ("EXT2-fs: unsupported inode size: %d\n",
				sb->u.ext2_sb.s_inode_size);
			goto failed_mount;
		}
	}
	sb->u.ext2_sb.s_frag_size = EXT2_MIN_FRAG_SIZE <<
				le32_to_cpu(es->s_log_frag_size);
	if (sb->u.ext2_sb.s_frag_size)
		sb->u.ext2_sb.s_frags_per_block = sb->s_blocksize /
					sb->u.ext2_sb.s_frag_size;
	else
		sb->s_magic = 0;
	sb->u.ext2_sb.s_blocks_per_group = le32_to_cpu(es->s_blocks_
		per_group);
	sb->u.ext2_sb.s_frags_per_group = le32_to_cpu(es->s_frags_
		per_group);
	sb->u.ext2_sb.s_inodes_per_group = le32_to_cpu(es->s_inodes_
		per_group);
	sb->u.ext2_sb.s_inodes_per_block = sb->s_blocksize /
				EXT2_INODE_SIZE(sb);
	sb->u.ext2_sb.s_itb_per_group=sb->u.ext2_sb.s_inodes_per_group/
				sb->u.ext2_sb.s_inodes_per_block;
	sb->u.ext2_sb.s_desc_per_block = sb->s_blocksize /
				sizeof (struct ext2_group_desc);
	sb->u.ext2_sb.s_sbh = bh;
	if (resuid != EXT2_DEF_RESUID)
		sb->u.ext2_sb.s_resuid = resuid;
	else
	sb->u.ext2_sb.s_resuid = le16_to_cpu(es->s_def_resuid);
	if (resgid != EXT2_DEF_RESGID)
		sb->u.ext2_sb.s_resgid = resgid;
```

```
else
sb->u.ext2_sb.s_resgid = le16_to_cpu(es->s_def_resgid);
sb->u.ext2_sb.s_mount_state = le16_to_cpu(es->s_state);
sb->u.ext2_sb.s_addr_per_block_bits =
      log2 (EXT2_ADDR_PER_BLOCK(sb));
sb->u.ext2_sb.s_desc_per_block_bits =
      log2 (EXT2_DESC_PER_BLOCK(sb));
if (sb->s_magic != EXT2_SUPER_MAGIC) {
      if (!silent)
      printk ("VFS: Can't find an ext2 filesystem on dev "
                  "%s.\n",
                  bdevname(dev));
      goto failed_mount;
}
if (sb->s_blocksize != bh->b_size) {
      if (!silent)
            printk ("VFS: Unsupported blocksize on dev "
                  "%s.\n", bdevname(dev));
      goto failed_mount;
}

if (sb->s_blocksize != sb->u.ext2_sb.s_frag_size) {
printk ("EXT2-fs: fragsize %lu != blocksize %lu (not
      supported yet)\n",
            sb->u.ext2_sb.s_frag_size, sb->s_blocksize);
      goto failed_mount;
}

if (sb->u.ext2_sb.s_blocks_per_group > sb->s_blocksize * 8) {
      printk ("EXT2-fs: #blocks per group too big: %lu\n",
            sb->u.ext2_sb.s_blocks_per_group);
      goto failed_mount;
}
if (sb->u.ext2_sb.s_frags_per_group > sb->s_blocksize * 8) {
printk ("EXT2-fs: #fragments per group too big: %lu\n",
            sb->u.ext2_sb.s_frags_per_group);
      goto failed_mount;
}
if (sb->u.ext2_sb.s_inodes_per_group > sb->s_blocksize * 8) {
```

```
        printk ("EXT2-fs: #inodes per group too big: %lu\n",
                sb->u.ext2_sb.s_inodes_per_group);
        goto failed_mount;
}

sb->u.ext2_sb.s_groups_count =(le32_to_cpu(es->s_blocks_count)-
                              le32_to_cpu(es->s_first_data_block)+
                              T2_BLOCKS_PER_GROUP(sb) - 1) /
                              XT2_BLOCKS_PER_GROUP(sb);
db_count = (sb->u.ext2_sb.s_groups_count+ EXT2_DESC_PER_
        BLOCK(sb)-1)/EXT2_DESC_PER_BLOCK(sb);
sb->u.ext2_sb.s_group_desc = kmalloc (db_count * sizeof
                        (struct buffer_head *), GFP_KERNEL);
if (sb->u.ext2_sb.s_group_desc == NULL) {
        printk ("EXT2-fs: not enough memory\n");
        goto failed_mount;
}
for (i = 0; i < db_count; i++) {
        sb->u.ext2_sb.s_group_desc[i] = bread (dev, logic_sb_
                        block+i+1,sb->s_blocksize);
        if (!sb->u.ext2_sb.s_group_desc[i]) {
                for (j = 0; j < i; j++)
                        brelse (sb->u.ext2_sb.s_group_desc[j]);
                kfree(sb->u.ext2_sb.s_group_desc);
printk ("EXT2-fs: unable to read group descriptors\n");
                goto failed_mount;
        }
}
if (!ext2_check_descriptors (sb)) {
        for (j = 0; j < db_count; j++)
                brelse (sb->u.ext2_sb.s_group_desc[j]);
        kfree(sb->u.ext2_sb.s_group_desc);
        printk ("EXT2-fs: group descriptors corrupted !\n");
        goto failed_mount;
}
for (i = 0; i < EXT2_MAX_GROUP_LOADED; i++) {
        sb->u.ext2_sb.s_inode_bitmap_number[i] = 0;
        sb->u.ext2_sb.s_inode_bitmap[i] = NULL;
        sb->u.ext2_sb.s_block_bitmap_number[i] = 0;
```

```
			sb->u.ext2_sb.s_block_bitmap[i] = NULL;
	}

	sb->u.ext2_sb.s_loaded_inode_bitmaps = 0;
	sb->u.ext2_sb.s_loaded_block_bitmaps = 0;
	sb->u.ext2_sb.s_gdb_count = db_count;
	/*
	 * set up enough so that it can read an inode
	 */
	sb->s_op = &ext2_sops;
	sb->s_root = d_alloc_root(iget(sb, EXT2_ROOT_INO));
	if (!sb->s_root) {
		for (i = 0; i < db_count; i++)
			if (sb->u.ext2_sb.s_group_desc[i])
				brelse (sb->u.ext2_sb.s_group_desc[i]);
		kfree(sb->u.ext2_sb.s_group_desc);
		brelse (bh);
		printk ("EXT2-fs: get root inode failed\n");
		return NULL;
	}
	ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
	return sb;
}

static void ext2_commit_super (struct super_block * sb,
			       struct ext2_super_block * es)
{
	es->s_wtime = cpu_to_le32(CURRENT_TIME);
	mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
	sb->s_dirt = 0;
}

/*
 * In the second extended file system, it is not necessary to
 * write the super block since we use a mapping of the
 * disk super block in a buffer.
 *
 * However, this function is still used to set the fs valid
 * flags to 0.  We need to set this flag to 0 since the fs
```

```
 * may have been checked while mounted and e2fsck may have
 * set s_state to EXT2_VALID_FS after some corrections.
 */

void ext2_write_super (struct super_block * sb)
{
      struct ext2_super_block * es;

      if (!(sb->s_flags & MS_RDONLY)) {
            es = sb->u.ext2_sb.s_es;

            ext2_debug ("setting valid to 0\n");

            if (le16_to_cpu(es->s_state) & EXT2_VALID_FS) {
      es->s_state = cpu_to_le16(le16_to_cpu(es->s_state) &
                  ~EXT2_VALID_FS);
                  es->s_mtime = cpu_to_le32(CURRENT_TIME);
            }
            ext2_commit_super (sb, es);
      }
      sb->s_dirt = 0;
}

int ext2_remount (struct super_block * sb, int * flags, char * data)
{
      struct ext2_super_block * es;
      unsigned short resuid = sb->u.ext2_sb.s_resuid;
      unsigned short resgid = sb->u.ext2_sb.s_resgid;
      unsigned long new_mount_opt;
      unsigned long tmp;

      /*
       * Allow the "check" option to be passed as a remount option.
       */
      new_mount_opt = sb->u.ext2_sb.s_mount_opt;
      if (!parse_options (data, &tmp, &resuid, &resgid, &new_mount_opt))
            return -EINVAL;

      sb->u.ext2_sb.s_mount_opt = new_mount_opt;
```

```
        sb->u.ext2_sb.s_resuid = resuid;
        sb->u.ext2_sb.s_resgid = resgid;
        es = sb->u.ext2_sb.s_es;
        if ((*flags & MS_RDONLY) == (sb->s_flags & MS_RDONLY))
              return 0;
        if (*flags & MS_RDONLY) {
              if (le16_to_cpu(es->s_state) & EXT2_VALID_FS ||
                  !(sb->u.ext2_sb.s_mount_state & EXT2_VALID_FS))
                    return 0;
              /*
         * OK, we are remounting a valid rw partition rdonly, so set
         * the rdonly flag and then mark the partition as valid again.
              */
        es->s_state = cpu_to_le16(sb->u.ext2_sb.s_mount_state);
              es->s_mtime = cpu_to_le32(CURRENT_TIME);
              mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
              sb->s_dirt = 1;
              ext2_commit_super (sb, es);
        }
        else {
              int ret;
              if ((ret = EXT2_HAS_RO_COMPAT_FEATURE(sb, EXT2_FEATURE_
                        RO_COMPAT_SUPP))) {
        printk("EXT2-fs: %s: couldn't remount RDWR because of "
                           "unsupported optional features (%x).\n",
                           bdevname(sb->s_dev), ret);
                    return -EROFS;
              }
              /*
         * Mounting a RDONLY partition read-write, so reread and
         * store the current valid flag.  (It may have been changed
         * by e2fsck since we originally mounted the partition.)
         */
        sb->u.ext2_sb.s_mount_state = le16_to_cpu(es->s_state);
              if (!ext2_setup_super (sb, es, 0))
                    sb->s_flags &= ~MS_RDONLY;
        }
        return 0;
}
```

```
int ext2_statfs (struct super_block * sb, struct statfs * buf)
{
      unsigned long overhead;
      int i;

      if (test_opt (sb, MINIX_DF))
            overhead = 0;
      else {
            /*
             * Compute the overhead (FS structures)
             */

            /*
             * All of the blocks before first_data_block are
             * overhead
             */
             overhead = le32_to_cpu(sb->u.ext2_sb.s_es->s_first_
                        data_block);

            /*
             * Add the overhead attributed to the superblock and
             * block group descriptors.  If the sparse superblocks
             * feature is turned on, then not all groups have this.
             */
            for (i = 0; i < EXT2_SB(sb)->s_groups_count; i++)
                  overhead += ext2_bg_has_super(sb, i) +
                        ext2_bg_num_gdb(sb, i);

            /*
             * Every block group has an inode bitmap, a block
             * bitmap, and an inode table.
             */
            overhead += (sb->u.ext2_sb.s_groups_count *
                       (2 + sb->u.ext2_sb.s_itb_per_group));
      }

      buf->f_type = EXT2_SUPER_MAGIC;
      buf->f_bsize = sb->s_blocksize;
      buf->f_blocks = le32_to_cpu(sb->u.ext2_sb.s_es->s_blocks_count)
```

```
                    - overhead;
      buf->f_bfree = ext2_count_free_blocks (sb);
      buf->f_bavail = buf->f_bfree - le32_to_cpu(sb->u.ext2_sb.s_
                    es->s_r_blocks_count);
      if (buf->f_bfree < le32_to_cpu(sb->u.ext2_sb.s_es->s_r_
                    blocks_count))
            buf->f_bavail = 0;
      buf->f_files = le32_to_cpu(sb->u.ext2_sb.s_es->s_inodes_count);
      buf->f_ffree = ext2_count_free_inodes (sb);
      buf->f_namelen = EXT2_NAME_LEN;
      return 0;
}

static DECLARE_FSTYPE_DEV(ext2_fs_type, "ext2", ext2_read_super);

static int __init init_ext2_fs(void)
{
       return register_filesystem(&ext2_fs_type);
}

static void __exit exit_ext2_fs(void)
{
      unregister_filesystem(&ext2_fs_type);
}

EXPORT_NO_SYMBOLS;

module_init(init_ext2_fs)
module_exit(exit_ext2_fs)
```

원천파일 fs/ext2/file.c(2.4.3)

```
/*
 *  linux/fs/ext2/file.c
 *
 * Copyright (C) 1992, 1993, 1994, 1995
 * Remy Card (card@masi.ibp.fr)
 * Laboratoire MASI - Institut Blaise Pascal
 * Universite Pierre et Marie Curie (Paris VI)
```

```
 *
 *  from
 *
 *  linux/fs/minix/file.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  ext2 fs regular file handling primitives
 *
 *  64-bit file support on 64-bit platforms by Jakub Jelinek
 *    (jj@sunsite.ms.mff.cuni.cz)
 */

#include <linux/fs.h>
#include <linux/ext2_fs.h>
#include <linux/sched.h>

/*
 * Called when an inode is released. Note that this is different
 * from ext2_file_open: open gets called at every open, but release
 * gets called only when /all/ the files are closed.
 */
static int ext2_release_file (struct inode * inode, struct file * filp)
{
      if (filp->f_mode & FMODE_WRITE)
            ext2_discard_prealloc (inode);
      return 0;
}
/*
 * We have mostly NULL's here: the current defaults are ok for
 * the ext2 filesystem.
 */
struct file_operations ext2_file_operations = {
      read:       generic_file_read,
      write:      generic_file_write,
      ioctl:      ext2_ioctl,
      mmap:       generic_file_mmap,
      open:       generic_file_open,
      release:    ext2_release_file,
```

```
	fsync:		ext2_sync_file,
};

struct inode_operations ext2_file_inode_operations = {
	truncate:	ext2_truncate,
};
```

fs/namei.c안의 함수 open_namei()의 원천코드

```
  int open_namei(const char * pathname, int flag, int mode, struct
nameidata *nd)
{
	int acc_mode, error = 0;
	struct inode *inode;
	struct dentry *dentry;
	struct dentry *dir;
	int count = 0;

	acc_mode = ACC_MODE(flag);

	/*
	 * The simplest case - just a plain lookup.
	 */
	if (!(flag & O_CREAT)) {
		if (path_init(pathname, lookup_flags(flag), nd))
			error = path_walk(pathname, nd);
		if (error)
			return error;
		dentry = nd->dentry;
		goto ok;
	}

	/*
	 * Create - we need to know the parent.
	 */
	if (path_init(pathname, LOOKUP_PARENT, nd))
		error = path_walk(pathname, nd);
	if (error)
```

```
		return error;

	/*
	 * We have the parent and last component. First of all, check
	 * that we are not asked to creat(2) an obvious directory - that
	 * will not do.
	 */
	error = -EISDIR;
	if (nd->last_type != LAST_NORM || nd->last.name[nd->last.len])
		goto exit;

	dir = nd->dentry;
	down(&dir->d_inode->i_sem);
	dentry = lookup_hash(&nd->last, nd->dentry);
do_last:
	error = PTR_ERR(dentry);
	if (IS_ERR(dentry)) {
		up(&dir->d_inode->i_sem);
		goto exit;
	}

	/* Negative dentry, just create the file */
	if (!dentry->d_inode) {
		error = vfs_create(dir->d_inode, dentry, mode);
		up(&dir->d_inode->i_sem);
		dput(nd->dentry);
		nd->dentry = dentry;
		if (error)
			goto exit;
		/* Don't check for write permission, don't truncate */
		acc_mode = 0;
		flag &= ~O_TRUNC;
		goto ok;
	}

	/*
	 * It already exists.
	 */
	up(&dir->d_inode->i_sem);
```

```
	error = -EEXIST;
	if (flag & O_EXCL)
		goto exit_dput;

	if (d_mountpoint(dentry)) {
		error = -ELOOP;
		if (flag & O_NOFOLLOW)
			goto exit_dput;
	while ( __follow_down(&nd->mnt,&dentry) && d_mountpoint
			dentry));
	}
	error = -ENOENT;
	if (!dentry->d_inode)
		goto exit_dput;
	if (dentry->d_inode->i_op && dentry->d_inode->i_op->follow_link)
		goto do_link;

	dput(nd->dentry);
	nd->dentry = dentry;
	error = -EISDIR;
	if (dentry->d_inode && S_ISDIR(dentry->d_inode->i_mode))
		goto exit;
ok:
	error = -ENOENT;
	inode = dentry->d_inode;
	if (!inode)
		goto exit;

	error = -ELOOP;
	if (S_ISLNK(inode->i_mode))
		goto exit;

	error = -EISDIR;
	if (S_ISDIR(inode->i_mode) && (flag & FMODE_WRITE))
		goto exit;

	error = permission(inode,acc_mode);
	if (error)
```

```
		goto exit;

	/*
	 * FIFO's, sockets and device files are special: they don't
	 * actually live on the filesystem itself, and as such you
	 * can write to them even if the filesystem is read-only.
	 */
	if (S_ISFIFO(inode->i_mode) || S_ISSOCK(inode->i_mode)) {
		flag &= ~O_TRUNC;
	} else if (S_ISBLK(inode->i_mode) || S_ISCHR(inode->i_mode)){
		error = -EACCES;
		if (IS_NODEV(inode))
			goto exit;

		flag &= ~O_TRUNC;
	} else {
		error = -EROFS;
		if (IS_RDONLY(inode) && (flag & 2))
			goto exit;
	}
	/*
	 * An append-only file must be opened in append mode for
		writing.
	 */
	error = -EPERM;
	if (IS_APPEND(inode)) {
		if  ((flag & FMODE_WRITE) && !(flag & O_APPEND))
			goto exit;
		if (flag & O_TRUNC)
			goto exit;
	}

	/*
	 * Ensure there are no outstanding leases on the file.
	 */
	error = get_lease(inode, flag);
	if (error)
		goto exit;
```

```
	if (flag & O_TRUNC) {
		error = get_write_access(inode);
		if (error)
			goto exit;

		/*
		 * Refuse to truncate files with mandatory locks held on
		   them.
		 */
		error = locks_verify_locked(inode);
		if (!error) {
			DQUOT_INIT(inode);

			error = do_truncate(dentry, 0);
		}
		put_write_access(inode);
		if (error)
			goto exit;
	} else
		if (flag & FMODE_WRITE)
			DQUOT_INIT(inode);

	return 0;

exit_dput:
	dput(dentry);
exit:
	path_release(nd);
	return error;

do_link:
	error = -ELOOP;
	if (flag & O_NOFOLLOW)
		goto exit_dput;
	/*
	 * This is subtle. Instead of calling do_follow_link() we do the
	 * thing by hands. The reason is that this way we have zero
link_count
```

```
         * and path_walk() (called from ->follow_link) honoring
LOOKUP_PARENT.
         * After that we have the parent and last component, i.e.
         * we are in the same situation as after the first path_walk().
         * Well, almost - if the last component is normal we get its copy
         * stored in nd->last.name and we will have to putname() it
           when we
         * are done. Procfs-like symlinks just set LAST_BIND.
         */
        UPDATE_ATIME(dentry->d_inode);
        error = dentry->d_inode->i_op->follow_link(dentry, nd);
        dput(dentry);
        if (error)
              return error;
        if (nd->last_type == LAST_BIND) {
              dentry = nd->dentry;
              goto ok;
        }
        error = -EISDIR;
        if (nd->last_type != LAST_NORM)
              goto exit;
        if (nd->last.name[nd->last.len]) {
              putname(nd->last.name);
              goto exit;
        }
        if (count++==32) {
              dentry = nd->dentry;
              putname(nd->last.name);
              goto ok;
        }
        dir = nd->dentry;
        down(&dir->d_inode->i_sem);
        dentry = lookup_hash(&nd->last, nd->dentry);
        putname(nd->last.name);
        goto do_last;
}
```

제5장. 론리기록권관리기 LVM

현재의 Linux봉사기에는 3~10개 또는 그이상의 디스크들이 장비되여 있다. SCSI통로와 분기된 디스크카비네트는 여러개의 디스크를 얼마든지 장비할수 있다. 그러나 여러개의 디스크구획(disk partition)을 관리하는 문제는 헐치 않다. 또한 Linux체계관리기들은 어떤 한개의 파일체계가 공간을 거의 100%차지하게 되자 구획을 아주 묘하게 확장할수 있는 방법을 모색하게 되였다. 이때 LVM을 리용하면 이 문제가 가능할뿐아니라 아주 쉽게 실현될수 있다.

LVM은 UNIX에서의 실현과정을 거쳐 사실상 표준기억관리형태로 된 직결디스크 기억관리용부분체계이다. LVM은 초기에 AIX조작체계용으로 IBM에 의하여 개발되였으며 계속하여 OSF/1조작체계용으로 OSF(현재 OpenGroup)에 의하여 리용되였다.

OSF판본은 그후 HP_UX와 Digital UNIX조작체계의 기초로 되였다. 이것이 바로 LVM들이 가동환경들에서 서로 류사한 리유로 된다. Linux에서 LVM은 HP_UX LVM과 아주 류사하다. 일반적으로 Linux LVM이 실현됨으로써 인터네트상에서의 리용률이 대단히 높아 졌다.

LVM은 파일체계와 기록권을 관리하는 방법을 완전히 새롭게 고찰한다. LVM은 구동기들이 현재 구획표도식을 리용하는것보다 더 유연한 방법으로 디스크들을 주사하고 크기를 재조직하며 관리할수 있게 한다.

LVM에 대한 소개

LVM은 핵심부에서 물리적장치들과 블로크 I/O대면부들사이에 보충적인 층을 추가한다. 실례로 ext2과 같은 파일체계는 디스크구동기를 직접 사용할 대신에 LVM에 의하여 제공된 블로크장치를 사용한다.

그림 5-1에 LVM에 의하여 도입된 I/O론리의 보충적인 계층을 제시하였다.

전통적인 체계들에서 디스크구동기들은 보통 련속적인 기억구역으로 구획화(《구획》 혹은 《단편》)되며 블로크장치로 넘겨 진다. PC체계상에서 구획은 fdisk와 같은 도구에 의하여 실현되며 이때 fdisk는 단순구획표를 보존하게 된다.

실례로 디스크구동기 /dev/sda는 *n*개의 기억구역으로 구획화되며 구획들은 그림 5-2에서 보여 준것처럼 /dev/sda*n*을 통하여 /dev/sda1과 련관되게 한다.

결함은 명백한바 구획은 크기가 디스크구동기의 크기로 제한되며 구획의 크기를 재설정하는 과정은 련속적인 구획들을 재구성하거나 혹은 여벌복사로 되돌아 가지 않고 디스크를 재조직하는 GNU parted*와 같은 특별한 도구를 사용할것을 요구한다는것이다. 이것은 아주 위험하고 모험적인 조작이다.

---

* 많은 독자들은 Power Quest Corp의 Partition Magic와 더 익숙되였을것이다.
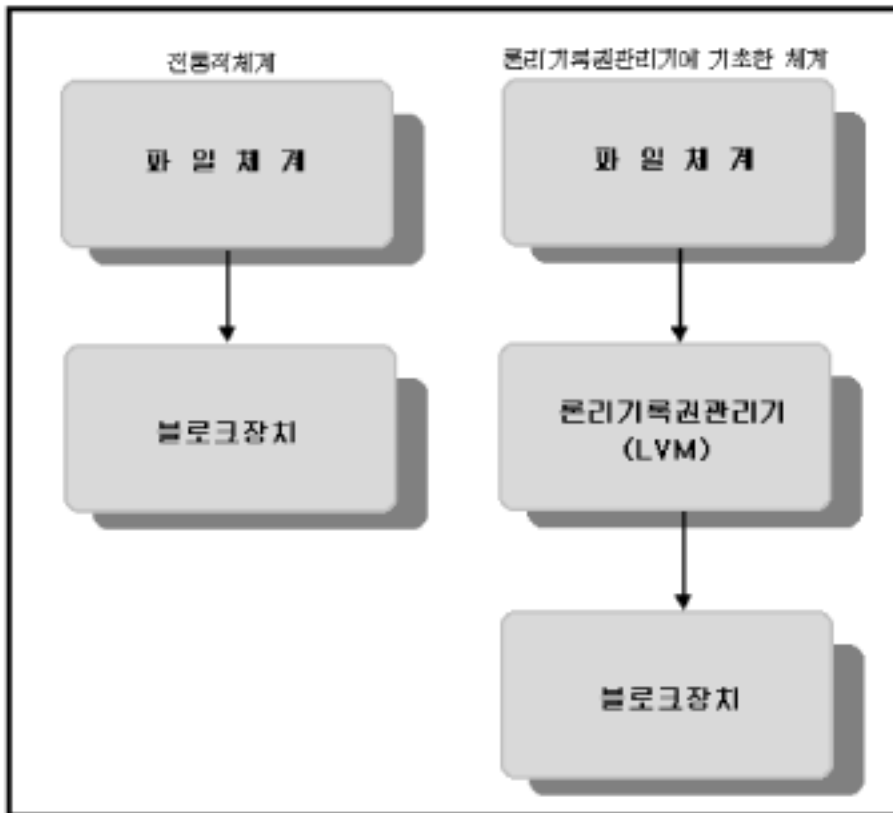
그림 5-1. 이전 블로크장치와 LVM에 기초한 블로크장치
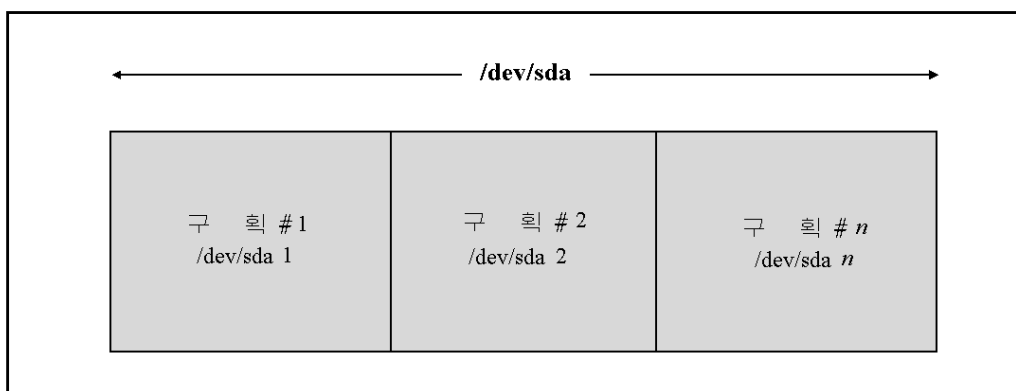
그림 5-2. 주어 진 디스크의 구획수

대조적으로 lvm블로크장치들은 물리적속박에 의하여 제한되지 않으며 련속되여 있을 필요도 없고 직결상태에서 크기를 재설정할수 있다. LVM체계는 한개 혹은 그이상의 물리기록권으로 구성되는 기록권그룹으로(VG 혹은 VOLG) 기억한다.

LVM블로크장치들은 론리기록권(LV)이라고 부르며 이 기록권들은 LVM에 의하여 보존되는 기억풀(pool)로부터 배정된다.

그림 5-3은 LVM환경에서 디스크들이 더이상 핵심부에 의하여 직접적으로 조종되지 않는다는것을 보여 준다.
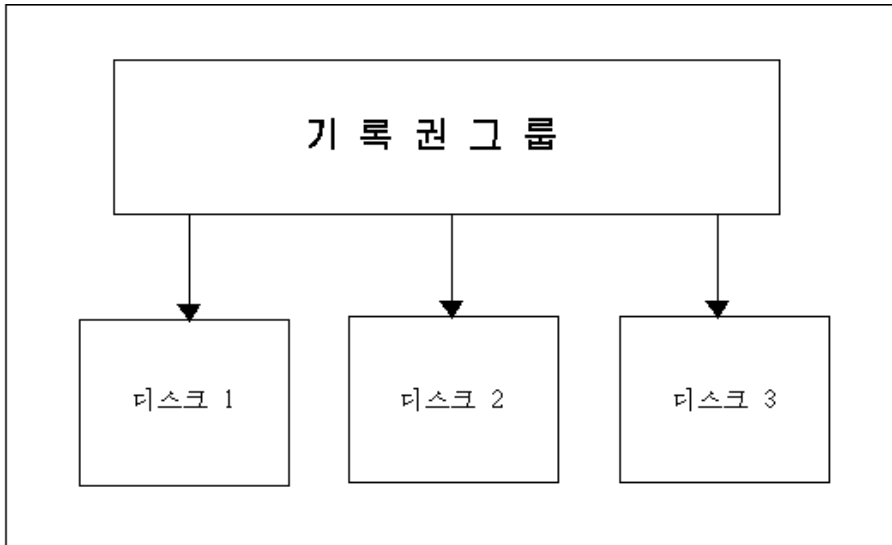
그림 5-3. 하나 혹은 그이상의 물리적디스크들로 구성된 기록권그룹

핵심부대신 LVM층이 디스크를 조종한다. 체계상에는 하나이상의 기록권그룹이 있을수 있다. 한번 만들어 지면 기록권그룹은 디스크가 아니며 자료를 기억하기 위한 기본단위로 된다(하나 혹은 그이상의 물리적디스크들로 구성되는 가상적디스크로 생각하면 된다.).

LVM은 여러개의 층으로 기억공간을 조직한다. 즉 기록권그룹은 하나 혹은 그이상의 다중디스크구동기의 용량과 결합된 풀이다. 디스크들이 고정크기의 련속된 구획들로 나누어 지는 현재의 구획도식과는 달리 LVM은 사용자가 디스크를 물리기록권으로서 즉 같은 크기의 범위들로 구성되는 자료기억의 풀(혹은 기록권)로서 고찰하게 한다. 기록권그룹에 의하여 표현되는 디스크공간의 풀은 여러가지 치수의 가상구획 혹은 론리기록권으로 나누어 질수 있다. 론리기록권은 여러 물리기록권으로 늘였다 줄였다 할수 있거나 혹은 한개 물리기록권의 한부분만을 표시할수도 있다. 론리기록권의 크기는 치수에 의하여 결정된다. 일단 생성되면 론리기록권은 파일체계의 생성이나 교환장치와 같이 정규디스크구획처럼 사용될수 있다.

LVM의 우점

론리기록권을 서로 다르게 만든다는것은 련속적으로 만들어 지지 않고 임의로 크기를 재설정할수 있다는것을 말한다. 만일 론리기록권상에 생성된 파일체계의 크기를 재설정할수 있으면 파일체계를 얼마든지 늘쿠거나 줄일수 있다.

더우기 LVM은 디스크자원들을 관리하기 위한 종합적인 관리틀을 가지고있다.

LVM은 핵심부의 블로크(디스크)장치대면부와 체계상의 실제적물리장치사이에 새로운 층을 놓는 방법으로 동작한다. 새로운 론리《장치》들은 하나 혹은 그이상의 물리적장치부분들을 리용하여 만들수 있다. 단순응용프로그람들은 다중구획(혹은 전체 구동기)을 단순하고 보다 큰 론리구동기들과 결합능력을 가진다.

하지만 LVM의 우점은 론리기록권의 치수를 재빨리 변화시킬수 있게 한다는데 있다. 구획이 너무 작아 지는 문제를 체험한 체계관리기는 《더 크게 할수 있다》고 말하였다. 체계를 지우거나 여벌복사, 재구획구성, 회복 등을 수행할 대신에 현재는 간단히 lvextend 명령을 줄수 있다.

LVM은 어떻게 동작하는가

LVM은 물리적주변장치들과 핵심부의 I/O대면부사이에 론리기록권의 보충적인 층 즉 새 론리층을 추가한다. 이 층은 사슬형식으로 결합된 여러개의 디스크(이른바 물리기록권)들이 물리범위라고 부르는(PE) 배정단위에 의하여 기억풀 혹은 기록권그룹을 형성하게 한다. 다음 기록권그룹외의 부분들은 론리범위라고 부르는 (LE)단위로 론리기록권의 형태로 배정될수 있다. 매 론리범위는 같은 크기의 대응하는 물리범위로 넘겨 진다. 론리기록권들은 /dev/VolumeGroupName/LogicalVolumeName로 명명되는 /dev/sd[a−z]* 혹은 /dev/ hd[a−z]*과 류사한 장치전용파일을 통하여 리용될수 있다.

매개 물리기록권, 기록권그룹, 론리기록권에 관한 배치구성정보는 기록권그룹서술자구역(Volume Group Descriptor Area 혹은 VGDA)이라고 부르는 구역의 물리기록권상에 기억된다. 이 정보를 쓸 때에 VGDA는 물리적으로 상위블로크의 바로 뒤에 위치하게 되는데 이것은 다음번 개방에서는 변경될수도 있다. 배치구성정보는 자동적으로 생성되는 여벌복사파일에 기억되며 이 파일은 /etc/lvmtab.d등록부에 기억된다.

LVM구동기는 론리기록권의 론리적범위와 물리기록권의 물리적범위사이의 넘기기표를 보존한다. 이 표들은 상위사용자 LVM명령들에 의하여 생성되고 갱신되며 또한 지워진다. 구동기의 기본넘기기함수는 /usr/src/linux/driver/block/ ll_rw_blk.c파일에 있는 ll_rw_block(), ll_rw_swap_file()함수들로부터 론리기록권에 있는 론리적블로크를 호출한다. 넘기기함수는 표안에서 대응하는 물리블로크/디스크쌍을 검색한다. 다음 디스크블로크들에 대한 물리적I/O요청이 대기렬로 형성되게 하는 ll_rw_*()함수에 이 쌍을 귀환시킨다. Linux에서 블로크장치는 단순한 구동기이며 이 구동기는 buffer_head들을 얻고 기억장치들로부터 자료를 채우든가 혹은 기억장치에 그 자료를 써넣는다.

특히 장치구동기가 서로 다른 장치 지어 여러개 장치들에 다른 I/O를 수행하여 I/O요청을 만족시킬수 있게 한다.

LVM과 쏘프트웨어RAID구동기는 하나 혹은 그이상의 I/O들을 론리기록권이나 RAID묶음에 속하는 하나 혹은 그이상의 디스크들에 관하여 실행시켜 가상장치에 대한 I/O를 실현하도록 하는 우점을 가지고 있다.

LVM 혹은 RAID구동기에 대한 단일I/O는 토대층디스크에 대한 단일I/O를 실현시킬 때 리용되는 특정한 최량화이다. 이 경우에 실제적으로 토대층디스크에 보내야 할 새로

운 I/O조작을 생성할 필요는 없다. 실제적으로 론리장치 즉 주어 진 물리장치가 기동시에 올려태우기된 블로크장치보다 다른 물리장치에 대하여 동작이 진행되여야 한다는것을 통보하기 위하여 론리장치는 읽혀 지거나 써지는 buffer_head에 표식자를 붙인다.

디스크나 구획(혹은 그것들이 새롭게 호출되는 기록권그룹/론리그룹)들은 장치의 가상적표현이기때문에 실행시에도 론리기록권을 쉽게 늘이거나 줄일수 있다.

이제 LVM이 이러한 기능들을 내부적으로 어떻게 실현하는가를 보자.

LVM의 내부

물리기록권이나 기록권그룹(volg) 그리고 론리기록권(lvol)을 생성할 때 이것들의 매 론리적실체에 관한 배치구성정보는 대응하는 물리기록권에 기억되며 이 정보의 여벌복사는 /etc/lvmtab.d등록부밑에 자동적으로 주어 진다.

핵심부2.3.49로부터 시작하여 LVM구동기는 핵심부안에 내장되여 있다. 이 구동기들은 volg와 론리기록권 그리고 대응하는 물리구획과 구동기사이의 넘기기표를 가지고 있다. 이 표들은 LVM명령들을 생성하고 유지하는데 이 LVM명령들은 뿌리만이 실행할수 있다.

LVM관리용lvol상의 한개 파일체계에 포함되여 있는 한개의 블로크에 대한 접근이 요구될 때마다 구동기는 해당 블로크를 그 디스크상의 실제주소로 넘기기하는데 이 표를 리용한다.

넘기기표의 리용

우리가 4개의 디스크를 포함하는 체계를 가지고 있다고 하자. 한개는 조작체계용이고 다른 3개는 자료용이다. LVM은 현재 4개의 디스크가 있다는것을 알고 있다. 왜냐하면 핵심부가 기동시에 디스크들을 검출하며 또 디스크목록으로 된 표를 보존하고 있기때문이다. 이제 해야 할것은 매개 디스크에 대하여 하나의 기록권그룹을 생성하는것이다. 매 파일체계에 대하여 필요하다면 개별적론리기록권을 계속 생성할수도 있다.

/dev/volgol/lvolroot for the OS	/800MB#this is one of two lvols for the 2GB disk used
/dev/volgol/lvolusr	/usr 1200MB # and this is the seond,for /usr
/dev/volgo2/lvolhome	/home 9000MB #
/dev/volgo3/lvoldbf	/ora_data 9000MB # this might actually be a RAID device
/dev/volgo4/lvolidx	/ora_idx

이제 우리가 어떤 공간밖에서 례를 들어 /home등록부밖에서 실행하며 사용자들이 굉장히 큰 MP3등록부라든지 동화상파일들중에서 어떤것을 지울수 없다고 가정하자. Linux에서는 아직 재치 있게 디스크공간을 생성할수 없기때문에 다른 디스크를 얻어야 할것이다. 현재 디스크들은 대다수 18GB에 도달하고 있다. 이제 새 그룹에 대하여 /dev/volgo2기록권디스크의 4GB구획을 지정한다. 그러면 LVM을 리용하여 최대 13GB로 lvolhome론리기록권을 확장하여 나간다(명백히 volgo2를 더 주지 않은 조건에서). 만약 현재 봉사기가 가속-교환디스크추가기능을 지원한다면 재기동할 필요가 없으며 /home은 곧 커지게 된다. 이때 일련의 경고가 있을수 있다.

첫째로 /home등록부로부터 사용자들을 순간적으로 분리시켜야 한다. 핵심부는 파일들이 그안에 아직 열려 져 있으면 /home등록부의 올려태우기해제를 하지 않게 된다. 대표적으로 봉사기상에서 모든 망동작(사용등록가입 등)을 순간동안 정지시키기 위하여 콤퓨터를 단일사용자방식으로 설정하고 자기에게 필요한 실행준위 (runlevel)로 돌아 간다.

둘째로 LVM을 리용하여 필요할 때마다 공간을 추가해야 한다. 그리고 사용자들은 인차 지워 지지 않는 등록부들에 모든 종류의 불필요한것들을 넣을수 있다.

몇가지 명령실례들

핵심부2.3.49+를 가지고 있거나 혹은 현재 핵심부에 LVM으로 검사수정한다고 할때 우에서 언급된 내용들을 수행하기 위하여 다음의 명령에 대한 실례들을 실행시킬수 있다.

1. LVM을 설치한후 "lnsmod lvm"을 수행하거나 혹은 그것을 자동적으로 적재하기 위하여 kerneld/kmod를 설정한다(INSTALL을 볼것).
2. 구획형Ox8e로 두개의 디스크상에 구획(#1)을 설정한다.
3. "pvcreate/dev/sd[ce]1"을 수행한다. 검사를 목적으로 하나의 디스크상에 한개이상의 1차구획이나 혹은 확장된 구획을 리용할수 있다.
4. "vgcreate volg02/dev/sd[ce]1"을 수행한다(vgcreate도 역시 기록권그룹을 활성화한다.).
5. 100MB의 선형론리기록권을 얻기 위하여서는 "lvcreate−lvoldbf_lv tvolg02"을 수행하거나 혹은 두개의 구획분할과 4KB의구획분할크기를 가진 1000MB이상의 큰 론리기록권을 얻기 위하여 "lvcreate−i2−14−l1000−nlvoldbf volg02"를 수행한다.
6. 요구될 때 생성된 LV를 리용한다. 실례로 "mke2fs/dev/volg02/lvoldbf"로 파일체계를 생성하고 그것을 올려 붙인다.

Linux LVM은 HP_UX개념 즉 명령들과 밀접히 동반되기때문에 이름과 동작에서 거의 같다. 그러므로 물리기록권을 조종하기 위한 명령들은 모두 pv로 시작하며 론리그룹조종을 위한 명령들은 vg로, 론리기록권을 조종하기 위한 명령들은 lv로 시작한다.

표-2 LVM명령들

명 령	기 능
2fsadm resizing for lvextend, lvreduce, e2fsck and resize2fs	파일체계를 포함하는 론리기록권을 위한 관리계층
lvchange	론리기록권의 속성변화
lvcreate	론리기록권의 생성
lvdisplay	론리기록권구성자료 현시
lvextend	론리기록권의 치수확장
lvreduce	론리기록권의 치수감소
lvrename	비사용론리기록권의 이름바꾸기
Lvscan	존재하는 모든 론리기록권찾기
lvmchange	LVM의 속성변화를 위한 긴급프로그람
lvmdiskscan	모든 디스크/구획, 다중장치와 그것들의 목록주사
lvmsadc	정적자료수집자
lvmsar	정적자료보고자
pvchange	물리기록권의 속성변화
pvcreate	물리기록권생성
pvdata	물리기록권그룹서술자구역목록의 디버그
pvdisplay	물리기록권배치구성의 연시
pvmove	론리적크기를 다른 물리기록권으로 이동
pvscan	존재하는 모든 물리기록권찾기
vgcfgbackup	모든 기록권그룹서술자구역의 여벌복사
vgcfgrestore	기록권그룹서술자구역의 디스크로의 회복
vgchange	능동/비능동기록권그룹
vgck	일관성을 위한 기록권그룹서술자구역의 검사
vgcreate	물리기록권으로부터 기록권그룹생성
vgdisplay	기록권그룹배치구성정보 연시
vgexport	기록권그룹내보내기(체계에 알려 지지 않게)
vgextend	하나 혹은 그이상의 물리기록권에 의하여 기록권그룹확장
vgimport	기록권그룹의 입수(체계 혹은 다른 체계에 알려 지게)
vgmerge	두개 기록권그룹을 하나로 련결
vgmknodes	모든 론리기록권전문에 의한 기록권그룹등록부생성
vgreduce	하나 혹은 그이상의 자유물리기록권에 의한 기록권그룹축소
vgremove	빈 기록권그룹제거
vgrename	비능동기록권그룹의 이름바꾸기
vgscan	기록권그룹에 대한 주사
vgsplit	한개 기록권그룹을 두개로 분할

LVM용물리기록권생성

LVM은 전체 물리기록권들이 기록권그룹으로 지적될것을 요구하므로 LVM이 리용할 수 있게 준비된 몇개의 빈 구획을 가지고 있어야 한다. 몇개의 구획상에 OS를 설치하고 빈 공간을 좀 남겨 둔다. 같은 크기를 가진 몇개의 빈 구획을 생성하기 위하여 Linux상에서 fdisk를 리용한다. fdisk를 리용하여 그 구획들에 형 OxFE로 표식을 달아야 한다. 이렇게 하여 5개의 256MB의 구획 /dev/hda5~/dev/hda9를 생성한다.

물리기록권의 등록

LVM을 실행하는데 필요한 첫번째 과제는 LVM에 물리기록권을 등록하는것이다. 이 과정은 pvcreate명령을 통하여 수행된다. 만들려는 매개 hdxx장치에 대하여 pvcreate/dev/hdxx을 간단히 실행하면 된다. 우리의 경우에는 실레로 pvcreate/dev/hda5 등으로 실행한다.

기록권그룹의 생성

다음 기록권그룹을 만든다. 기록권그룹생성명령에 물리적범위의 크기와 같은 어떤 인수들을 설정할수는 있으나 대체로 기정값을 리용하는것이 좋다. 새로운 기록권그룹 vgol을 호출한다. 즉 vgcreate/vgol/dev/hda5를 건입력한다. 이 명령이 수행된 다음 vgdisplay명령으로 기록권그룹을 본다. 여기서 256개까지 론리기록권을 생성할수 있고 256개까지 물리기록권을 추가할수 있으며 매개 론리기록권은 255.99GB까지 될수 있다는데 대하여 주의해야 한다. 보다 중요하게는 빈 물리적범위(PE)렬에 대한것이다. 이것은 론리기록권들을 생성할 때 작업할수 있는 물리적범위가 몇개나 될수 있는가를 말해 준다. 256MB디스크에 대하여 이 값은 크기가 4GB인 물리적범위보다 작은 쓰지 못하는 나머지가 존재하기때문에 63으로 된다.

론리기록권의 생성

다음으로 Vg vgol이라고 부르는 론리기록권을 만들어 보자. 론리기록권을 만들 때 변경시켜도 일 없는 몇가지 설정항목이 존재하지만 역시 기정값을 쓰는것이 좋다. 생성하는데서 중요한 선택항목(option)은 해당 론리기록권우에 론리적범위를 몇개 배정하겠는가 하는것이다. 이제 총 16MB의 크기에 대하여 4개를 가지고 출발하겠다.

Lvcreate−14−nlvol vgol을 건입력한다.

−l대신 −L을 리용하여 MB단위로 크기를 정의할수도 있으며 이때 LVM은 가장 가까운 론리적범위의 크기중복은 둥그리기하여 잘라 버린다.

이제 lvdisplay명령으로 lvdisplay −v/dev/vgol/lvol을 건입력하여 론리기록권을 볼수 있다. 이제부터는 론리범위페지를 무시할수도 있고 보다 흥미 있는 자료들을 보기 위하여 페지를 우로 올릴수도 있다.

기록권그룹에 디스크의 추가

다음으로 /dev/hdab을 기록권그룹에 추가하겠다.

vgectend vgol/dev/hda을 건입력하여 수행한다.

이것을 vgdisplay−v vgol을 리용하여 검사할수 있다. 보다 더 많은 PE(물리범위)들이 존재할수 있다는것을 강조한다.

구획분할(striped)된 론리기록권생성

LVM이 기록권그룹안의 한개 물리기록권에 대하여 옹근 하나의 론리기록권을 생성했다는것을 강조한다. Lvcreate명령에 −l기발을 설정하여 lv를 두개의 물리기록권으로 구획분할할수 있다. 이제 새로운 론리기록권lvo2를 생성하고 hda5와 hda6으로 구획분할하겠다.

lvcreate−I4−nlvo2−i2 vgo1/dev/hda5/dev/hda6을 건입력한다.

명령선상에서의 물리기록권의 정의는 LVM이 어느 물리적범위를 리용하겠는가를 지적하며 한편 −i2명령은 두개의 기록권으로 구획분할한다는것을 지적한다. 따라서 현재 우리는 두개의 물리기록권으로 구획이 분할된 한개의 론리기록권을 가지게 된다.

기록권그룹안에서 자료의 이동

지금까지 물리적범위와 론리적범위는 거의 호환적이였다. 이것들은 크기도 같고 또 LVM에 의하여 자동적으로 넘기기된다. 하지만 이것은 사실상 그렇지 않다. 디스크가 올려태우기되여 리용중에 있어도 하나의 pv로부터 다른것에로 전체 lv를 옮길수 있다. 이것은 체계성능에는 영향을 주지만 쓸모 있게 개선할수 있다.

hda5로부터 hda6으로 lvol을 옮기자.

Pvmove −n/dev/vgol/lvol/dev/hda5/dev/hda6을 건입력한다. 이 명령은 /dev/hda5의 PE들에 넘기기된 lvo1이 리용하는 모든 LE들을 /dev/hda6의 새로운 PE들로 옮긴다. 이 과정에 따라 hda5로부터 hda6으로 효과적으로 자료가 옮겨 진다. 조작실행후 lvdisplay −v/dev/vgol/lvol로 결과를 볼수 있으며 자료가 총체적으로 /dev/hda6에 존재한다는것을 알수 있다.

기록권그룹으로부터 론리기록권제거

이제 더이상 lv02이 필요없다고 하자. 이런 경우 우리는 그것을 제거할수 있으며 기록권그룹을 위한 빈 풀(pool)에 론리기록권들의 PE들을 돌려 보낼수 있다. 우선 그것의 파일체계를 올려태우기해제한다. 다음 lvchange −a n/dev/vg0l/lv02를 리용하여 그것을 비능동상태로 만든다. 마지막으로 lvremove/dev/vg0l/lv02를 건입력하여 지운다. 기록권그룹을 보고 PE들이 현재 사용되지 않는다는것을 알수 있다.

기록권그룹으로부터 디스크의 제거

기록권그룹으로부터 디스크를 역시 제거할수 있다. 이제 hda5를 더이상 리용하지 않으면 그것을 기록권그룹으로부터 제거할수 있다.

Vgreduce vg01/dev/hda5를 건입력하면 제거동작이 실현된다.

원천코드 include/linux/lvm.h

```
/*
/* include/linux/lvm.h
 * kernel/lvm.h
 * tools/lib/lvm.h
 *
 * Copyright (C) 1997 - 2000  Heinz Mauelshagen, Sistina Software
 *
 * February-November 1997
 * May-July 1998
 * January-March,July,September,October,Dezember 1999
 * January,February,July,November 2000
 * January-March 2001
 *
 * lvm is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * lvm is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU CC; see the file COPYING.  If not, write to
 * the Free Software Foundation, 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 *
 */

/*
 * Changelog
 *
 *    10/10/1997 - beginning of new structure creation
 *    12/05/1998 - incorporated structures from lvm_v1.h and deleted
       lvm_v1.h
 *    07/06/1998 - avoided LVM_KMALLOC_MAX define by using
```

```
 vmalloc/vfree
*             instead of kmalloc/kfree
*   01/07/1998 - fixed wrong LVM_MAX_SIZE
*   07/07/1998 - extended pe_t structure by ios member (for
 statistic)
*   02/08/1998 - changes for official char/block major numbers
*   07/08/1998 - avoided init_module() and cleanup_module() to be
 static
*   29/08/1998 - separated core and disk structure type definitions
*   01/09/1998 - merged kernel integration version (mike)
*   20/01/1999 - added LVM_PE_DISK_OFFSET macro for use in
*             vg_read_with_pv_and_lv(), pv_move_pe(),
                  pv_show_pe_text()...
*   18/02/1999 - added definition of time_disk_t structure for;
*             keeps time stamps on disk for nonatomic writes
                  (future)
*   15/03/1999 - corrected LV() and VG() macro definition to use
                  argument
*             instead of minor
*   03/07/1999 - define for genhd.c name handling
*   23/07/1999 - implemented snapshot part
*   08/12/1999 - changed LVM_LV_SIZE_MAX macro to reflect current
                  1TB limit
*   01/01/2000 - extended lv_v2 core structure by wait_queue member
*   12/02/2000 - integrated Andrea Arcagnelli's snapshot work
*   18/02/2000 - seperated user and kernel space parts by
*             #ifdef them with __KERNEL__
*   08/03/2000 - implemented cluster/shared bits for vg_access
*   26/06/2000 - implemented snapshot persistency and resizing support
*   02/11/2000 - added hash table size member to lv structure
*   12/11/2000 - removed unneeded timestamp definitions
*   24/12/2000 - removed LVM_TO_{CORE,DISK}*, use cpu_{from, to}_le*
*             instead - Christoph Hellwig
*   22/01/2001 - Change ulong to uint32_t
*   14/02/2001 - changed LVM_SNAPSHOT_MIN_CHUNK to 1 page
*   20/02/2001 - incremented IOP version to 11 because of incompatible
*             change in VG activation (in order to support
                  devfs better)
*   01/03/2001 - Revert to IOP10 and add VG_CREATE_OLD call for
```

```
                    compatibility
 *    08/03/2001 - new lv_t (in core) version number 5: changed page
                    member
 *               to (struct kiobuf *) to use for COW exception
                    table io
 *    26/03/2001 - changed lv_v4 to lv_v5 in structure definition (HM)
 *
 */
#ifndef _LVM_H_INCLUDE
#define _LVM_H_INCLUDE

#define LVM_RELEASE_NAME "0.9.1_beta7"
#define LVM_RELEASE_DATE "10/04/2001"

#define    _LVM_KERNEL_H_VERSION  "LVM "LVM_RELEASE_NAME"
("LVM_RELEASE_DATE")"

#include <linux/version.h>

/*
 * preprocessor definitions
 */
/* if you like emergency reset code in the driver */
#define    LVM_TOTAL_RESET

#ifdef __KERNEL__
#undef LVM_HD_NAME /* display nice names in /proc/partitions */

/* lots of debugging output (see driver source)
  #define DEBUG_LVM_GET_INFO
  #define DEBUG
  #define DEBUG_MAP
  #define DEBUG_MAP_SIZE
  #define DEBUG_IOCTL
  #define DEBUG_READ
  #define DEBUG_GENDISK
  #define DEBUG_VG_CREATE
  #define DEBUG_DEVICE
  #define DEBUG_KFREE
```

```
 */
#endif                              /* #ifdef __KERNEL__ */

#ifndef __KERNEL__
#define __KERNEL__
#include <linux/kdev_t.h>
#include <linux/list.h>
#undef __KERNEL__
#else
#include <linux/kdev_t.h>
#include <linux/list.h>
#endif                              /* #ifndef __KERNEL__ */

#include <asm/types.h>
#include <linux/major.h>

#ifdef __KERNEL__
#if Linux_VERSION_CODE >= KERNEL_VERSION ( 2, 3 ,0)
#include <linux/spinlock.h>
#else
#include <asm/spinlock.h>
#endif

#include <asm/semaphore.h>
#endif                              /* #ifdef __KERNEL__ */

#include <asm/page.h>

#if !defined ( LVM_BLK_MAJOR) || !defined ( LVM_CHAR_MAJOR)
#error Bad include/linux/major.h - LVM MAJOR undefined
#endif

#ifdef      BLOCK_SIZE
#undef      BLOCK_SIZE
#endif

#ifdef CONFIG_ARCH_S390
#define BLOCK_SIZE     4096
#else
```

```
#define BLOCK_SIZE     512
#endif

#ifndef     SECTOR_SIZE
#define SECTOR_SIZE    512
#endif

#define LVM_STRUCT_VERSION   1     /* structure version */

#define     LVM_DIR_PREFIX   "/dev/"

#ifndef min
#define min(a,b) (((a)<(b))?(a):(b))
#endif
#ifndef max
#define max(a,b) (((a)>(b))?(a):(b))
#endif

/* set the default structure version */
#if ( LVM_STRUCT_VERSION == 1)
#define pv_t pv_v2_t
#define lv_t lv_v5_t
#define vg_t vg_v3_t
#define pv_disk_t pv_disk_v2_t
#define lv_disk_t lv_disk_v3_t
#define vg_disk_t vg_disk_v2_t
#define lv_block_exception_t lv_block_exception_v1_t
#define lv_COW_table_disk_t lv_COW_table_disk_v1_t
#endif

/*
 * i/o protocol version
 *
 * defined here for the driver and defined seperate in the
 * user land tools/lib/liblvm.h
 *
 */
#define     LVM_DRIVER_IOP_VERSION        10
```

```
#define LVM_NAME        "lvm"
#define LVM_GLOBAL      "global"
#define LVM_DIR         "lvm"
#define LVM_VG_SUBDIR  "VGs"
#define LVM_LV_SUBDIR  "LVs"
#define LVM_PV_SUBDIR  "PVs"

/*
 * VG/LV indexing macros
 */
/* character minor maps directly to volume group */
#define    VG_CHR(a) ( a)

/* block minor indexes into a volume group/logical volume indirection
table */
#define    VG_BLK(a)   ( vg_lv_map[a].vg_number)
#define LV_BLK(a)( vg_lv_map[a].lv_number)

/*
 * absolute limits for VGs, PVs per VG and LVs per VG
 */
#define ABS_MAX_VG     99
#define ABS_MAX_PV     256
#define ABS_MAX_LV     256   /* caused by 8 bit minor */

#define MAX_VG  ABS_MAX_VG
#define MAX_LV   ABS_MAX_LV
#define    MAX_PV     ABS_MAX_PV

#if ( MAX_VG > ABS_MAX_VG)
#undef MAX_VG
#define MAX_VG ABS_MAX_VG
#endif

#if ( MAX_LV > ABS_MAX_LV)
#undef MAX_LV
#define MAX_LV ABS_MAX_LV
#endif
```

```
/*
 * VGDA: default disk spaces and offsets
 *
 *   there's space after the structures for later extensions.
 *
 *   offset                       what                        size
 *---------------   ----------------------------    ------------
 *   0                   physical volume structure          ~500 byte
 *
 *   1K                  volume group structure             ~200 byte
 *
 *   6K                  namelist of physical volumes       128 byte each
 *
 *   6k+n*~300byte       n logical volume structures        ~300 byte each
 *
 *   + m * 4byte       m physical extent alloc. structs     4 byte each
 *
 *   End of disk -       first physical extent    typically   4 megabyte
 *   PE total *
 *   PE size
 *
 *
 */

/* DONT TOUCH THESE !!! */
/* base of PV structure in disk partition */
#define     LVM_PV_DISK_BASE       0L

/* size reserved for PV structure on disk */
#define     LVM_PV_DISK_SIZE       1024L

/* base of VG structure in disk partition */
#define     LVM_VG_DISK_BASE       LVM_PV_DISK_SIZE

/* size reserved for VG structure */
#define     LVM_VG_DISK_SIZE       ( 9 * 512L)
/* size reserved for timekeeping */
#define     LVM_TIMESTAMP_DISK_BASE      ( LVM_VG_DISK_BASE +
LVM_VG_DISK_SIZE)
```

```
#define     LVM_TIMESTAMP_DISK_SIZE      512L  /* reserved for
timekeeping */

/* name list of physical volumes on disk */
#define     LVM_PV_UUIDLIST_DISK_BASE (LVM_TIMESTAMP_DISK_BASE + \
                               LVM_TIMESTAMP_DISK_SIZE)

/* now for the dynamically calculated parts of the VGDA */
#define     LVM_LV_DISK_OFFSET(a, b) ( (a)->lv_on_disk.base + \
                              sizeof ( lv_disk_t) * b)
#define     LVM_DISK_SIZE(pv)       ( (pv)->pe_on_disk.base + \
                              (pv)->pe_on_disk.size)
#define     LVM_PE_DISK_OFFSET(pe, pv)   ( pe * pv->pe_size + \
                               ( LVM_DISK_SIZE ( pv) / SECTOR_SIZE))
#define     LVM_PE_ON_DISK_BASE(pv) \
   { int rest; \
     pv->pe_on_disk.base = pv->lv_on_disk.base + pv->lv_on_disk.size; \
     if ( ( rest = pv->pe_on_disk.base % SECTOR_SIZE) != 0) \
        pv->pe_on_disk.base += ( SECTOR_SIZE - rest); \
   }
/* END default disk spaces and offsets for PVs */


/*
 * LVM_PE_T_MAX corresponds to:
 *
 * 8KB PE size can map a ~512 MB logical volume at the cost of 1MB memory,
 *
 * 128MB PE size can map a 8TB logical volume at the same cost of memory.
 *
 * Default PE size of 4 MB gives a maximum logical volume size of 256 GB.
 *
 * Maximum PE size of 16GB gives a maximum logical volume size of 1024 TB.
 *
 * AFAIK, the actual kernels limit this to 1 TB.
 *
 * Should be a sufficient spectrum ;*)
 */
```

```
/* This is the usable size of pe_disk_t.le_num !!!        v     v */
#define     LVM_PE_T_MAX      ( ( 1 << ( sizeof ( uint16_t) * 8)) - 2)

#define     LVM_LV_SIZE_MAX(a)     ( ( long long) LVM_PE_T_MAX * (a)-
>pe_size > ( long long) 1024*1024/SECTOR_SIZE*1024*1024 ? ( long long)
1024*1024/SECTOR_SIZE*1024*1024 : ( long long) LVM_PE_T_MAX * (a)-
>pe_size)
#define     LVM_MIN_PE_SIZE        ( 8192L / SECTOR_SIZE) /* 8 KB in
sectors */
#define     LVM_MAX_PE_SIZE        ( 16L * 1024L * 1024L / SECTOR_SIZE *
1024) /* 16GB in sectors */
#define     LVM_DEFAULT_PE_SIZE    ( 4096L * 1024 / SECTOR_SIZE)/* 4 MB
in sectors */
#define     LVM_DEFAULT_STRIPE_SIZE      16L   /* 16 KB  */
#define     LVM_MIN_STRIPE_SIZE    ( PAGE_SIZE/SECTOR_SIZE)     /*
PAGESIZE in sectors */
#define     LVM_MAX_STRIPE_SIZE    ( 512L * 1024 / SECTOR_SIZE) /* 512 KB
in sectors */
#define     LVM_MAX_STRIPES        128   /* max # of stripes */
#define     LVM_MAX_SIZE          ( 1024LU * 1024 / SECTOR_SIZE * 1024 *
1024) /* 1TB[sectors] */
#define     LVM_MAX_MIRRORS        2     /* future use */
#define     LVM_MIN_READ_AHEAD     2     /* minimum read ahead sectors */
#define     LVM_MAX_READ_AHEAD     120   /* maximum read ahead sectors */
#define     LVM_MAX_LV_IO_TIMEOUT  60    /* seconds I/O timeout (future
use) */
#define     LVM_PARTITION         0xfe  /* LVM partition id */
#define     LVM_NEW_PARTITION      0x8e  /* new LVM partition id
(10/09/1999) */
#define     LVM_PE_SIZE_PV_SIZE_REL      5     /* max relation PV size and
PE size */

#define     LVM_SNAPSHOT_MAX_CHUNK 1024  /* 1024 KB */
#define     LVM_SNAPSHOT_DEF_CHUNK 64    /* 64  KB */
#define     LVM_SNAPSHOT_MIN_CHUNK (PAGE_SIZE/1024) /* 4 or 8 KB */

#define     UNDEF -1
#define     FALSE  0
#define     TRUE   1
```

```
#define LVM_GET_COW_TABLE_CHUNKS_PER_PE(vg, lv) ( \
      vg->pe_size / lv->lv_chunk_size)

#define LVM_GET_COW_TABLE_ENTRIES_PER_PE(vg, lv) ( \
{ \
      int COW_table_entries_per_PE; \
      int COW_table_chunks_per_PE; \
\
      COW_table_entries_per_PE = LVM_GET_COW_TABLE_CHUNKS_PER_PE(vg,
lv); \
      COW_table_chunks_per_PE = ( COW_table_entries_per_PE *
sizeof(lv_COW_table_disk_t) / SECTOR_SIZE + lv->lv_chunk_size - 1) / lv-
>lv_chunk_size; \
      COW_table_entries_per_PE - COW_table_chunks_per_PE;})

/*
 * ioctls
 */
/* volume group */
#define    VG_CREATE_OLD         _IOW ( 0xfe, 0x00, 1)
#define    VG_REMOVE             _IOW ( 0xfe, 0x01, 1)

#define    VG_EXTEND             _IOW ( 0xfe, 0x03, 1)
#define    VG_REDUCE             _IOW ( 0xfe, 0x04, 1)

#define    VG_STATUS              _IOWR ( 0xfe, 0x05, 1)
#define    VG_STATUS_GET_COUNT    _IOWR ( 0xfe, 0x06, 1)
#define    VG_STATUS_GET_NAMELIST  _IOWR ( 0xfe, 0x07, 1)

#define    VG_SET_EXTENDABLE      _IOW ( 0xfe, 0x08, 1)
#define    VG_RENAME              _IOW ( 0xfe, 0x09, 1)

/* Since 0.9beta6 */
#define    VG_CREATE             _IOW ( 0xfe, 0x0a, 1)

/* logical volume */
#define    LV_CREATE             _IOW ( 0xfe, 0x20, 1)
#define    LV_REMOVE             _IOW ( 0xfe, 0x21, 1)
```

```
#define     LV_ACTIVATE            _IO ( 0xfe, 0x22)
#define     LV_DEACTIVATE          _IO ( 0xfe, 0x23)

#define     LV_EXTEND             _IOW ( 0xfe, 0x24, 1)
#define     LV_REDUCE             _IOW ( 0xfe, 0x25, 1)

#define     LV_STATUS_BYNAME       _IOWR ( 0xfe, 0x26, 1)
#define     LV_STATUS_BYINDEX      _IOWR ( 0xfe, 0x27, 1)

#define LV_SET_ACCESS              _IOW ( 0xfe, 0x28, 1)
#define LV_SET_ALLOCATION          _IOW ( 0xfe, 0x29, 1)
#define LV_SET_STATUS              _IOW ( 0xfe, 0x2a, 1)

#define LE_REMAP                   _IOW ( 0xfe, 0x2b, 1)

#define LV_SNAPSHOT_USE_RATE       _IOWR ( 0xfe, 0x2c, 1)

#define     LV_STATUS_BYDEV        _IOWR ( 0xfe, 0x2e, 1)

#define     LV_RENAME              _IOW ( 0xfe, 0x2f, 1)

#define     LV_BMAP                _IOWR ( 0xfe, 0x30, 1)

/* physical volume */
#define     PV_STATUS             _IOWR ( 0xfe, 0x40, 1)
#define     PV_CHANGE             _IOWR ( 0xfe, 0x41, 1)
#define     PV_FLUSH              _IOW ( 0xfe, 0x42, 1)

/* physical extent */
#define     PE_LOCK_UNLOCK         _IOW ( 0xfe, 0x50, 1)

/* i/o protocol version */
#define     LVM_GET_IOP_VERSION    _IOR ( 0xfe, 0x98, 1)

#ifdef LVM_TOTAL_RESET
/* special reset function for testing purposes */
#define     LVM_RESET             _IO ( 0xfe, 0x99)
#endif
```

```
/* lock the logical volume manager */
#define     LVM_LOCK_LVM            _IO ( 0xfe, 0x100)
/* END ioctls */

/*
 * Status flags
 */
/* volume group */
#define     VG_ACTIVE            0x01       /* vg_status */
#define     VG_EXPORTED          0x02       /*    "    */
#define     VG_EXTENDABLE        0x04       /*    "    */

#define     VG_READ              0x01       /* vg_access */
#define     VG_WRITE             0x02       /*    "    */
#define     VG_CLUSTERED         0x04       /*    "    */
#define     VG_SHARED            0x08       /*    "    */

/* logical volume */
#define     LV_ACTIVE            0x01       /* lv_status */
#define     LV_SPINDOWN          0x02       /*    "    */

#define     LV_READ              0x01       /* lv_access */
#define     LV_WRITE             0x02 /*     "     */
#define     LV_SNAPSHOT          0x04 /*     "     */
#define     LV_SNAPSHOT_ORG      0x08 /*     "     */

#define     LV_BADBLOCK_ON       0x01 /*    lv_badblock */

#define     LV_STRICT            0x01 /*     lv_allocation */
#define     LV_CONTIGUOUS        0x02 /*       "       */

/* physical volume */
#define     PV_ACTIVE            0x01       /* pv_status */
#define     PV_ALLOCATABLE       0x02       /* pv_allocatable */

/* misc */
#define LVM_SNAPSHOT_DROPPED_SECTOR 1
```

```
/*
 * Structure definitions core/disk follow
 *
 * conditional conversion takes place on big endian architectures
 * in functions * pv_copy_*(), vg_copy_*() and lv_copy_*()
 *
 */

#define     NAME_LEN         128   /* don't change!!! */
#define     UUID_LEN         32    /* don't change!!! */

/* copy on write tables in disk format */
typedef struct {
      uint64_t pv_org_number;
      uint64_t pv_org_rsector;
      uint64_t pv_snap_number;
      uint64_t pv_snap_rsector;
} lv_COW_table_disk_v1_t;

/* remap physical sector/rdev pairs including hash */
typedef struct {
      struct list_head hash;
      uint32_t rsector_org;
      kdev_t   rdev_org;
      uint32_t rsector_new;
      kdev_t   rdev_new;
} lv_block_exception_v1_t;

/* disk stored pe information */
typedef struct {
      uint16_t lv_num;
      uint16_t le_num;
} pe_disk_t;

/* disk stored PV, VG, LV and PE size and offset information */
typedef struct {
      uint32_t base;
```

```
	uint32_t size;
} lvm_disk_data_t;

/*
 * Structure Physical Volume (PV) Version 1
 */

/* core */
typedef struct {
	char id[2];		/* Identifier */
	unsigned short version;	/* HM lvm version */
	lvm_disk_data_t pv_on_disk;
	lvm_disk_data_t vg_on_disk;
	lvm_disk_data_t pv_namelist_on_disk;
	lvm_disk_data_t lv_on_disk;
	lvm_disk_data_t pe_on_disk;
	char pv_name[NAME_LEN];
	char vg_name[NAME_LEN];
	char system_id[NAME_LEN];	/* for vgexport/vgimport */
	kdev_t pv_dev;
	uint pv_number;
	uint pv_status;
	uint pv_allocatable;
	uint pv_size;		/* HM */
	uint lv_cur;
	uint pe_size;
	uint pe_total;
	uint pe_allocated;
	uint pe_stale;		/* for future use */
	pe_disk_t *pe;		/* HM */
	struct inode *inode;	/* HM */
} pv_v1_t;

/* core */
typedef struct {
	char id[2];		/* Identifier */
	unsigned short version;/* HM lvm version */
```

```
	lvm_disk_data_t pv_on_disk;
	lvm_disk_data_t vg_on_disk;
	lvm_disk_data_t pv_uuidlist_on_disk;
	lvm_disk_data_t lv_on_disk;
	lvm_disk_data_t pe_on_disk;
	char pv_name[NAME_LEN];
	char vg_name[NAME_LEN];
	char system_id[NAME_LEN];	/* for vgexport/vgimport */
	kdev_t pv_dev;
	uint pv_number;
	uint pv_status;
	uint pv_allocatable;
	uint pv_size;		/* HM */
	uint lv_cur;
	uint pe_size;
	uint pe_total;
	uint pe_allocated;
	uint pe_stale;		/* for future use */
	pe_disk_t *pe;		/* HM */
	struct inode *inode;	/* HM */
	char pv_uuid[UUID_LEN+1];
} pv_v2_t;

/* disk */
typedef struct {
	uint8_t id[2];		/* Identifier */
	uint16_t version;	/* HM lvm version */
	lvm_disk_data_t pv_on_disk;
	lvm_disk_data_t vg_on_disk;
	lvm_disk_data_t pv_namelist_on_disk;
	lvm_disk_data_t lv_on_disk;
	lvm_disk_data_t pe_on_disk;
	uint8_t pv_name[NAME_LEN];
	uint8_t vg_name[NAME_LEN];
	uint8_t system_id[NAME_LEN]; /* for vgexport/vgimport */
	uint32_t pv_major;
	uint32_t pv_number;
```

```
	uint32_t pv_status;
	uint32_t pv_allocatable;
	uint32_t pv_size;	/* HM */
	uint32_t lv_cur;
	uint32_t pe_size;
	uint32_t pe_total;
	uint32_t pe_allocated;
} pv_disk_v1_t;

/* disk */
typedef struct {
	uint8_t id[2];		/* Identifier */
	uint16_t version;	/* HM lvm version */
	lvm_disk_data_t pv_on_disk;
	lvm_disk_data_t vg_on_disk;
	lvm_disk_data_t pv_uuidlist_on_disk;
	lvm_disk_data_t lv_on_disk;
	lvm_disk_data_t pe_on_disk;
	uint8_t pv_uuid[NAME_LEN];
	uint8_t vg_name[NAME_LEN];
	uint8_t system_id[NAME_LEN]; /* for vgexport/vgimport */
	uint32_t pv_major;
	uint32_t pv_number;
	uint32_t pv_status;
	uint32_t pv_allocatable;
	uint32_t pv_size;	/* HM */
	uint32_t lv_cur;
	uint32_t pe_size;
	uint32_t pe_total;
	uint32_t pe_allocated;
} pv_disk_v2_t;

/*
 * Structures for Logical Volume (LV)
 */

/* core PE information */
```

```
typedef struct {
      kdev_t dev;
      uint32_t pe;            /* to be changed if > 2TB */
      uint32_t reads;
      uint32_t writes;
} pe_t;

typedef struct {
      char lv_name[NAME_LEN];
      kdev_t old_dev;
      kdev_t new_dev;
      uint32_t old_pe;
      uint32_t new_pe;
} le_remap_req_t;

typedef struct lv_bmap {
      uint32_t lv_block;
      dev_t lv_dev;
} lv_bmap_t;

/*
 * Structure Logical Volume (LV) Version 3
 */

/* core */
typedef struct lv_v5 {
      char lv_name[NAME_LEN];
      char vg_name[NAME_LEN];
      uint lv_access;
      uint lv_status;
      uint lv_open;           /* HM */
      kdev_t lv_dev;          /* HM */
      uint lv_number;         /* HM */
      uint lv_mirror_copies; /* for future use */
      uint lv_recovery;/*      "       */
      uint lv_schedule;/*      "       */
      uint lv_size;
```

```
	pe_t *lv_current_pe;	/* HM */
	uint lv_current_le;	/* for future use */
	uint lv_allocated_le;
	uint lv_stripes;
	uint lv_stripesize;
	uint lv_badblock;/* for future use */
	uint lv_allocation;
	uint lv_io_timeout;	/* for future use */
	uint lv_read_ahead;

	/* delta to version 1 starts here */
	struct lv_v5 *lv_snapshot_org;
	struct lv_v5 *lv_snapshot_prev;
	struct lv_v5 *lv_snapshot_next;
	lv_block_exception_t *lv_block_exception;
	uint lv_remap_ptr;
	uint lv_remap_end;
	uint lv_chunk_size;
	uint lv_snapshot_minor;
#ifdef __KERNEL__
	struct kiobuf *lv_iobuf;
	struct kiobuf *lv_COW_table_iobuf;
	struct semaphore lv_snapshot_sem;
	struct list_head *lv_snapshot_hash_table;
	uint32_t lv_snapshot_hash_table_size;
	uint32_t lv_snapshot_hash_mask;
#if Linux_VERSION_CODE > KERNEL_VERSION ( 2, 3, 0)
	wait_queue_head_t lv_snapshot_wait;
#else
	struct wait_queue *lv_snapshot_wait;
#endif
	int   lv_snapshot_use_rate;
	void  *vg;

	uint lv_allocated_snapshot_le;
#if Linux_VERSION_CODE < KERNEL_VERSION(2, 3, 0)
	struct buffer_head **bheads;
```

```
#endif
#else
      char dummy[200];
#endif
} lv_v5_t;

/* disk */
typedef struct {
      uint8_t lv_name[NAME_LEN];
      uint8_t vg_name[NAME_LEN];
      uint32_t lv_access;
      uint32_t lv_status;
      uint32_t lv_open;      /* HM */
      uint32_t lv_dev;       /* HM */
      uint32_t lv_number;    /* HM */
      uint32_t lv_mirror_copies;   /* for future use */
      uint32_t lv_recovery;  /*      "       */
      uint32_t lv_schedule;  /*      "       */
      uint32_t lv_size;
      uint32_t lv_snapshot_minor;/* minor number of original */
      uint16_t lv_chunk_size;      /* chunk size of snapshot */
      uint16_t dummy;
      uint32_t lv_allocated_le;
      uint32_t lv_stripes;
      uint32_t lv_stripesize;
      uint32_t lv_badblock;       /* for future use */
      uint32_t lv_allocation;
      uint32_t lv_io_timeout;      /* for future use */
      uint32_t lv_read_ahead;      /* HM */
} lv_disk_v3_t;

/*
 * Structure Volume Group (VG) Version 1
 */
/* core */
typedef struct {
      char vg_name[NAME_LEN];      /* volume group name */
```

```
    uint vg_number;        /* volume group number */
    uint vg_access;        /* read/write */
    uint vg_status;        /* active or not */
    uint lv_max;           /* maximum logical volumes */
    uint lv_cur;           /* current logical volumes */
    uint lv_open;          /* open    logical volumes */
    uint pv_max;           /* maximum physical volumes */
    uint pv_cur;           /* current physical volumes FU */
    uint pv_act;           /* active physical volumes */
    uint dummy;            /* was obsolete max_pe_per_pv */
    uint vgda;             /* volume group descriptor arrays FU */
    uint pe_size;          /* physical extent size in sectors */
    uint pe_total;         /* total of physical extents */
    uint pe_allocated;     /* allocated physical extents */
    uint pvg_total;        /* physical volume groups FU */
    struct proc_dir_entry *proc;
    pv_t *pv[ABS_MAX_PV + 1];    /* physical volume struct pointers */
    lv_t *lv[ABS_MAX_LV + 1];    /* logical  volume struct pointers */
} vg_v1_t;
typedef struct {
    char vg_name[NAME_LEN]; /* volume group name */
    uint vg_number;         /* volume group number */
    uint vg_access;         /* read/write */
    uint vg_status;         /* active or not */
    uint lv_max;            /* maximum logical volumes */
    uint lv_cur;            /* current logical volumes */
    uint lv_open;           /* open    logical volumes */
    uint pv_max;            /* maximum physical volumes */
    uint pv_cur;            /* current physical volumes FU */
    uint pv_act;            /* active physical volumes */
    uint dummy;             /* was obsolete max_pe_per_pv */
    uint vgda;              /* volume group descriptor arrays FU */
    uint pe_size;           /* physical extent size in sectors */
    uint pe_total;          /* total of physical extents */
    uint pe_allocated;      /* allocated physical extents */
    uint pvg_total;         /* physical volume groups FU */
    struct proc_dir_entry *proc;
```

```
	pv_t *pv[ABS_MAX_PV + 1];	/* physical volume struct pointers */
	lv_t *lv[ABS_MAX_LV + 1];	/* logical  volume struct pointers */
	char vg_uuid[UUID_LEN+1];	/* volume group UUID */
#ifdef __KERNEL__
	struct proc_dir_entry *vg_dir_pde;
	struct proc_dir_entry *lv_subdir_pde;
	struct proc_dir_entry *pv_subdir_pde;
#else
	char dummy1[200];
#endif
} vg_v3_t;

/* disk */
typedef struct {
	uint8_t vg_name[NAME_LEN];	/* volume group name */
	uint32_t vg_number;		/* volume group number */
	uint32_t vg_access;		/* read/write */
	uint32_t vg_status;		/* active or not */
	uint32_t lv_max;		/* maximum logical volumes */
	uint32_t lv_cur;		/* current logical volumes */
	uint32_t lv_open;		/* open    logical volumes */
	uint32_t pv_max;		/* maximum physical volumes */
	uint32_t pv_cur;		/* current physical volumes FU */
	uint32_t pv_act;		/* active physical volumes */
	uint32_t dummy;
	uint32_t vgda;			/* volume group descriptor arrays FU */
	uint32_t pe_size;		/* physical extent size in sectors */
	uint32_t pe_total;		/* total of physical extents */
	uint32_t pe_allocated;		/* allocated physical extents */
	uint32_t pvg_total;		/* physical volume groups FU */
} vg_disk_v1_t;

typedef struct {
	uint8_t vg_uuid[UUID_LEN];	/* volume group UUID */
	uint8_t vg_name_dummy[NAME_LEN-UUID_LEN]; /* rest of v1 VG name */
	uint32_t vg_number;		/* volume group number */
	uint32_t vg_access;		/* read/write */
```

```
	uint32_t vg_status;    /* active or not */
	uint32_t lv_max;       /* maximum logical volumes */
	uint32_t lv_cur;       /* current logical volumes */
	uint32_t lv_open;      /* open    logical volumes */
	uint32_t pv_max;       /* maximum physical volumes */
	uint32_t pv_cur;       /* current physical volumes FU */
	uint32_t pv_act;       /* active physical volumes */
	uint32_t dummy;
	uint32_t vgda;         /* volume group descriptor arrays FU */
	uint32_t pe_size;      /* physical extent size in sectors */
	uint32_t pe_total;     /* total of physical extents */
	uint32_t pe_allocated; /* allocated physical extents */
	uint32_t pvg_total;    /* physical volume groups FU */
} vg_disk_v2_t;

/*
 * Request structures for ioctls
 */

/* Request structure PV_STATUS_BY_NAME... */
typedef struct {
	char pv_name[NAME_LEN];
	pv_t *pv;
} pv_status_req_t, pv_change_req_t;

/* Request structure PV_FLUSH */
typedef struct {
	char pv_name[NAME_LEN];
	kdev_t pv_dev;
} pv_flush_req_t;

/* Request structure PE_MOVE */
typedef struct {
	enum {
		LOCK_PE, UNLOCK_PE
	} lock;
	struct {
```

```
            kdev_t lv_dev;
            kdev_t pv_dev;
            uint32_t pv_offset;
      } data;
} pe_lock_req_t;

/* Request structure LV_STATUS_BYNAME */
typedef struct {
      char lv_name[NAME_LEN];
      lv_t *lv;
} lv_status_byname_req_t, lv_req_t;

/* Request structure LV_STATUS_BYINDEX */
typedef struct {
      uint32_t lv_index;
      lv_t *lv;
      /* Transfer size because user space and kernel space differ */
      ushort size;
} lv_status_byindex_req_t;

/* Request structure LV_STATUS_BYDEV... */
typedef struct {
      dev_t dev;
      lv_t *lv;
} lv_status_bydev_req_t;

/* Request structure LV_SNAPSHOT_USE_RATE */
typedef struct {
      int   block;
      int   rate;
} lv_snapshot_use_rate_req_t;

#endif                       /* #ifndef _LVM_H_INCLUDE */
```

제6장. Linux용 RAID

현재 Linux용파일체계의 조종하에 굉장히 많은 자료가 RAID디스크장치에 기억되여 있다. 따라서 Linux가 RAID장치를 어떻게 관리하는가에 대한 총체적인 리해를 가지는것은 아주 중요하다.

RAID의 기술적실현문제는 이 책에 밝혀 져 있지 않기때문에 RAID를 어떻게 제대로 리용하겠는가에 대하여 주의를 돌리겠다. 이 장은 RAID를 실제적으로 설치하는것과 또 Linux핵심부의 현재 배포판을 리용하여 RAID (Redundant Array of Inexpensive/Independent Disk)를 배치구성하는것이 얼마나 쉬운가를 보여 주게 된다(이 책의 경우 2.4.0).

인텔PC장치상에서 RAID를 실현하는 방법에는 3가지가 있다. 가장 공통적인 방법은 PCI SCSI RAID조종장치의 리용이다. Linux상에서 이 방법을 리용하는데서 문제는 고속말단(고속-사용자)이 많고 프로그람작성정보를 얻기 위하여 NDA를 요구하는것이다. 이 NDA는 원천코드를 개방할수 없는것으로 하여 무료쏘프트웨어를 금지하고 있다.

Linux하에서 RAID를 실현하는 공통적인 방법의 다른 하나는 SCSI대SCSI RAID조종장치를 리용하는것이다. 이 방법은 SCSI를 지원하는 조종장치를 요구한다(이것들중에는 여러가지가 있다.).

그러한 조종장치의 모선상에는 RAID조종장치가 있으며 이것은 하나 혹은 그이상의 장치들일수 있다(조종장치에 배렬을 어떻게 설정하는가에 따라).

RAID조종장치는 배렬을 포함하는 물리적장치들과 련결된 자체의 SCSI모선을 가지고 있다.

우리는 RAID배렬을 설정하기 위하여 임의의 블로크지원장치(IDE디스크, SCSI 등)를 리용할수 있다. RAID의 모든 조작들은 핵심부의 스레드들에 의하여 조종된다. 이 스레드들은 Linux핵심부들로부터 완성된 형태로 리용할수 있을것이다.

쏘프트웨어RAID는 관리응용프로그람들과 함께 핵심부모듈들의 모임이며 이때 관리응용프로그람들은 순수 프로그람적으로 RAID를 실현하며 특정한 장치를 요구하지 않는다.

Linux의 RAID부분체계는 저준위디스크구동기(IDE, SCSI, paraport구동기 등)와 블로크장치대면부우에 놓여 있는 핵심부의 한개 층으로 실현된다.

ext2fs, DOS-FAT 등의 파일체계는 블로크장치대면부우에 놓인다. 자체의 고유한 속성에 의하여 쏘프트웨어RAID도 장치적해결책보다 더 유연한 해결책을 지향한다. 부족점은 대등한 장치체계보다 CPU주기가 더 많이 요구되고 실행에 필요한 전력도 더 많이 요구된다는데 있다.

쏘프트웨어RAID는 보다 중요한 하나의 구별되는 특성을 가지고 있다. 그것은 구획대 구획에 기초하여 동작하며 여기서 여러개의 개별적디스크구획들은 RAID구획을 생성하기 위하여 서로 동시작용한다. 이 방법은 전체 디스크구동기를 한개 배렬로 동시동작시키는 대다수의 장치RAID방법과는 대조적이다.

장치적해결방식으로서의 RAID배렬이 있다는 사실은 관리를 단순화하려는 목적을 가

진 조작체계에 대해서는 명백하다. 프로그람적해결방식에 의거하면 보다 많은 선택성을 가지지만 조작이 복잡해 진다.

PCI조종장치

현재 Linux에서 리용할수 있거나 지원되는 장치에는 두개의 PCI SCSI RAID조종장치 즉 DPT와 ICP-vortex가 있다. Linux개발자들도 역시 Mylex카드의 지원하에서 작업하고 있으나 그것은 아직 완성되지 않은 상태에 있다.

DPT는 현재 몇년동안 지원되고 있는데 문제점은 배치구성프로그람을 실행할수 있는 DOS나 SCO를 요구한다는데 있다. Linux에 이 프로그람들을 이식하기 위한 약속은 되여 있으나 아직 실현되지 않고 있다. DPT는 필요한 정보를 제공하는 방법으로 Linux를 지원하고 있는데 실제작업은 거의 Michael Neuffer에 의하여 진행되고 있다.

DPT는 다중통로조종장치들과 캐쉬장치를 포함하는 몇가지 좋은 특성을 가지고 있다. 여기에는 카드상에 들을수 있는 경보기능뿐만아니라 외부프로쎄스에 리용할수 있는 구동기통지문도 있다. 조종프로그람은 없지만 RAID배렬조종을 위한 간단한 프로그람과 관리기가 써넣기 쉬운 e-mail도 있다.

ICP-Vortex는 Linux에 의하여 완전히 지원되며 배치구성을 위하여 그 어떤 다른 OS를 요구하지 않는다. 모든 ICP-Vortex의 배치구성은 카드의 BIOS준위에서 실현한다. 만일 배렬안의 임의의 디스크들에 문제가 생길 때 체계관리기를 감시하는 Linux데몬(묵인수속)이 있다. ICP는 카드의 배렬을 리용할수 있는 여러개의 모형을 가지고 있는데 이 카드들은 오직 RAID0과 1로부터 다중통로 RAID5카드들로 구성될수 있다(지어 빛섬유통로도).

RAID0/1카드는 만약 RAID5에 대한 요구가 제기되면 RAID5의 능력을 만족시킬수 있는 프로그람을 통하여 갱신할수 있다. 모든 카드들은 72핀 SIMM소케트를 거쳐 캐쉬를 추가할수 있는 능력을 가지고 있다(EDO와 표준50ns SIMMs는 둘다 잘 동작한다.).

ICP-Vortex는 오직 판본 2.0.33에서만 지원되는 부족점이 있다.

SCSI 대 SCSI조종장치

SCSI 대 SCSI해결방법은 여러가지로 제기되고 있다. 례하면 Mylex, CMD 그리고 다른 업체를 포함하는 많은 회사들로부터 해결방법들이 제기되고 있다. 이 방법들은 보통 외장시키는 방법이며 조종장치의 케스안에 완전한 슬로트를 요구한다. 실제적관리부는 말단모방기나 직렬포구를 거쳐 전면에 누르개단추들이 달려 있는 LCD판을 통하여 실현된다. 어떤 모형들은 말단모방기는 직렬포구중의 하나만을 가지며 또 어떤것들은 둘다 가지고 있다. 사용자들은 조종장치자체에 디스크배렬을 설정하며 조종장치는 디스크배렬을 한개의 론리장치로 OS에 준다(어떻게 설정하는가에 따라 한개이상의 론리장치일수도 있다.).

일반적으로 이 형태의 조종장치들이 가지고 있는 가장 큰 약점은 가격이 매우 비싼것이다.

CMD조종장치들은 Linux프로그람이 관리를 목적으로 하는 장치들중의 하나에 불과하다. 이 장치들은 2중여유림시교환조종장치를 가진것을 포함하여 리용가능한 여러가지 모형을 가지고 있다. 만약 현재 있는 하나의 조종장치에 고장이 있어도 봉사기를 멈춰세우지 않고 그것을 교체할수 있다. CMD조종장치는 또한 확장카드를 리용하여 다중통로로 확장갱신할수도 있다. 많은 개발자들이 SCSI 대 SCSI해결방법들을 제공하는데 여기서 취급한 내용은 오직 Mylex를 가지고만 해본것이다.

쏘프트웨어 RAID

최종적인 해결방법은 핵심부의 쏘프트웨어 RAID이다. RAID 0과 1은 핵심부에 도입된지 퍽 오래 되였으며 검사수정들은 지금 RAID 0~5를 지원하는 2.0.X핵심부에 리용되고 있는데 현재 2.2핵심부에서 표준으로 되고 있다. 쏘프트웨어 RAID는 장치체계전반에 대하여 매우 빨리 검사할수 있도록 개선되였다. 또한 임의의 블로크지원장치로 사용할수 있게 되여 있다. 이것은 한개의 디스크배렬에 IDE나 SCSI와 같은 장치들을 서로 섞어 리용할수 있다는것을 의미하는데 현존 장치수준으로는 이것이 완전히 불가능하다(사용자들이 요구한다는것은 의심할바 없지만 적어도 한개 장치를 분실하였거나 교체할것이 IDE 장치 하나만일 때에 도움이 될수 있다.).

기본부족점은 봉사기가 RAID의 기우성계산보다도 다른 함수계산에 CPU주기들을 필요로 한다는데 있다. 쏘프트웨어RAID의 가장 큰 우점은 가격이 눅은것이다.

쏘프트웨어RAID는 현재도 역시 하나의 문제점을 가지고 있는데 그것은 뿌리파일체계에 대하여서만 RAID를 제공하는것이다. 이 문제는 체계를 기동시킬수 있는 기동플로피디스크나 혹은 작은 RAID비기동구획을 리용하여 극복할수 있다.

하지만 그것을 실제적장애물로는 보지 않는다. 임의의 봉사기는 어떤 방법으로 리용할수 있는 플로피디스크를 가지고 있으며 이 플로피디스크는 값이 눅고 못 쓰게 되는 경우에 쉽게 바꿀수 있으므로 응용프로그람용의 완전한 기동매체로 되고 있다. 일단 2.2핵심부가 리용가능하게 되면(그리고 안정한 상태) 사용자는 기동플로피디스크나 소규모기동구획 혹은 어떤 다른 형식의 제거가능한 매체(Zip구동기, CD−ROM 등)를 리용하여 체계를 설치할 때 RAID지원기능을 요구한다.

여러가지 RAID장치들은 단일한 실패가능성을 제거하여 일정한 여유를 가지고 다양한 방법으로 배치구성된다. 배치구성은 장치적인 가동환경, 조작체계, 기억관리프로그람 혹은 특별한 장치들에 의하여 설정될수 있다. Linux의 경우에는 이 작업이 구동기들을 통하여 실현된다.

사용자는 실현성과 성능을 높일수 있도록 혼합된 배치구성방식으로 RAID를 리용할수 있다. LlNUX2.2.X가 나온 이래 RAID구동기들은 모든 표준배포판들에 배포되였다.

RAID의 기술에는 여러가지 준위가 존재하는데 다음의것들은 성능과 실현성, 리용성이 좋은것들이다.

RAID0 여기서는 리용성이 기본이 아니며 디스크가 RAID0으로 배치구성되는데 이때는 파일들속에 I/O를 련결할수 있도록 디스크를 통해 파일을 구획분할할 여유가 없다.

RAID1 RAID준위 1은 보호를 제공하는 첫번째 준위이다. 이 준위는 사용자가 두개 혹은 그이상의 디스크들을 《거울》형식으로 쓸수 있게 하며 따라서 자료는 하나이상의 디스크전체에 걸쳐 재생되게 된다. 만일 한개 디스크가 고장나면 현재 리용되는 RAID체계는 고장난 디스크를 쉽게 피하고 동작하는 디스크들만 리용한다. 두개의 디스크들에 대한 RAID1리용공간은 그 디스크들중의 하나의 크기와 같다(이것은 두 디스크가 같은 크기를 가지는 경우이고 만약 서로 크기가 다르면 리용할수 있는 공간은 정확히 두 디스크들중에서 작은것과 같다.). RAID체계는 배렬의 모든 디스크사이에서 균형읽기가 가능하므로 읽기상태에서 RAID1의 성능은 좋아 진다. 하지만 쓰기성능은 그리 좋지 못하다. 왜냐하면 전체 디스크에 대하여 쓰기가 생겨야 하기때문이다. 이러한 성능은 디스크들에서 중요하게 읽기를 중시하기때문에(사실상 전체 시간의 95%정도 높다.) 많은 체계(주로 파일봉사기형체계)들에서는 좋은 점으로 된다.

RAID0+1 RAID0의 구획분할과 RAID1을 1대1거울로 결합한다. 따라서 이 기술은 고속성과 리용성을 다같이 높인다.

RAID3 배렬안의 한개 디스크에 대한 기우성정보를 기억하기 위한 여유도를 제공한다. 이 기우성정보는 파괴된 다른 디스크상의 자료를 포함시키는데 도움을 줄수 있다. RAID3은 RAID1과 비교되는 정도의 기억을 디스크상에 보존하지만 기우성디스크가 병목(bottleneck)현상에 빠질수 있기때문에 자주 쓰지는 않는다.

RAID5 RAID3과 류사한 여유도로 기우성자료를 리용하지만 실제적인 자료를 구획분할하는 방법과 류사하게 모든 디스크에 걸쳐 기우성자료를 구획분할한다. 이 방식은 기우성디스크상에서 병목현상을 완화시킨다. RAID5는 아주 흔히 리용된다. RAID준위 5는 자료리용성과 성능을 둘다 가장 좋게 절충시킨다. RAID5는 사용자가 적어도 3개의 디스크배렬을 생성할수 있게 한다. 공간의 량은 일반적으로 전체 디스크총량−1에 의하여 계산된다. 례를 들어 4개의 9GB디스크를 리용할 때 사용가능한 공간의 크기는 3×9GB 즉 27GB로 된다. 작은 여유무효공간이 있을수 있으나 그것은 제한되여 있다. RAID5에 의하여 자료는 전체 디스크를 구획분할한 양식에 기억된다. 자그마한 차이가 있다면 역시 기우성정보도 자료와 함께 구획분할된다는것이다. 따라서 어떤 디스크가 자기의 내용을 잃어 버리게 되면 아직 리용할수 있는 자료와 기우성자료를 합하는 방법으로 재생성할수 있게 구획분할화가 리용되기때문에 성능은 일반적으로 좋다. 쓰기성능은 기우성발생과 기록용여유자료로 인하여 표준보다는 좀 뜨다. 읽기성능은 읽기가 디스크들에 걸쳐 균형화되므로 좋다. 여러 디스크에 대하여 읽기할 때와 자료기지화된 자료나 기우성자료를 발생시키는 경우에

재생성되여야 할 자료와 같이 배렬이 디스크를 놓치는 경우에는 성능이 현저히 떨어 진다. 그러나 결정적인 사실은 전체 디스크들이 모두 기능이 정지된다고 해도 자료는 여전히 살아 있다는것이다.

RAID6을 비롯하여 보충적인 RAID준위들이 더 있다.

RAID6은 2중기우성자료를 추가한것이며 RAID7과 RAID8은 RAID5의 특성에 강화된 성능을 보충한것들이다.

자료기지를 포함하는 봉사기환경이나 직결거래처리(OLTP)에서는 RAID0+1과 RAID5가 혼합된 배치구성을 흔히 볼수 있다. Linux에서는 실제적으로 배치구성이 매우 쉽다. Linux환경에서는 RAID를 실현하기 위하여 두개 파일 즉 /etc/raidtab와 /etc/rc.d/rc.local만을 편집하면 된다. 이 개념은 Linux가 RAID국부단위로 분할디스크구획에 접근하기 위한 특별한 구동기 즉 /dev/md0을 제공하는데 있다. 이 경우의 인상적인 점은 RAID환경에서는 분할구획들이 실제로 다른 디스크로 되여야 할 필요가 없다는것이다. 대다수의 경우에 Linux봉사기상에는 한개의 디스크만이 포함되여 있다. 새롭고 좋은 점은 우리가 이 장에서 배운것을 이제 실현해 볼수 있다는데도 있다.

RAID의 설정을 위한 단계를 목록으로 보여 준다.

▼ /dev/md0구동기를 배치구성한다.
- RAID환경에서 구획들을 초기화하고 그것에 대하여 /etc/raidtab에 통보한다.
- /etc/rc.d/rc.local에서 RAID동작을 자동적으로 실행한다.

▲ 올려태우기점아래에 RAID구동기를 올려태우기한다.

구획분할화(striping)

RAID를 리용하는 근거가 성능이라면 RAID0(구획분할화)을 실현해야 한다. 오직 한개의 하드디스크로 RAID를 사람들이 안전하게 실현해 볼수 있게 하기 위하여 여분의 구획 /dev/sda3과 /dev/sda4를 리용한다(이것은 우리가 SCSI디스크를 리용한다는것을 의미하며 그렇지 않은 경우에는 /dev/hda3과 /dev/hda4로 된다.).

분명히 생성환경에서는 이것이 동일한 디스크나 조종장치에 대하여 서로 다른 RAID구획을 준다는 감을 전혀 주지 않는다.

RAID0배치구성

RAID0을 배치구성하기 위하여 첫째로 /etc/raidtab에 RAID0환경으로 설정하려고 하는 구동기와 구획에 대하여 통보해야 한다.

/etc/raidtab 아래에서 다음과 같은것을 볼수 있다.

```
raiddev                dev/md0
   raid-level             0
```

```
nr-raid-disks           2
nr-spare-disks          0
chunk/size              4
persistent-superblock   1
device              /dev/sda3
raid-disk               0
device             /dev/sda4
raid-disk               1
```

일단 우의 파일들이 정확히 생성되면 새로운 RAID디스크들을 양식화하여야 한다(RAID0은 성능제고를 위하여 여러 디스크사이에 자료를 분배한다는데 대하여 기억해야 한다. 물론 이 실례에서는 두개의 구획이 다같은 물리적디스크에 있기때문에 성능상 리득은 없을것이다.).

RAID디스크를 양식화하기 위하여 다음의 조작을 실행한다(명백히 root에 관하여). 즉

```
mkraid/dev/md0.
```

다음 이 새로운 론리디스크상에 새로운 ext2fs파일체계를 만든다. 즉

```
mkfs -t ext2/dev/md0
```

마지막으로 기동시에 RAID구동기가 자동적으로 동작할수 있게 /etc/rc.d/rc.local에 한개 행을 보충하여야 한다. 즉

```
raidstart/dev/md0
```

이제 새로운 RAID론리구동기를 /opt2 선택으로 올려태우기하려고 한다고 가정하면 우와 같은 파일에 다음의 행을 추가한다(먼저행 다음에).

```
mount/dev/md0/opt2
```

왜 이 행을 보통 그것이 속하는 /etc/fstab에 설정하지 않는가?

실제적으로 제기될수 있는 문제점이다. Linux핵심부는 raidstart/dev/md0을 실행할 기회를 얻기전에 올려태우기하려고 하므로 실패하게 된다. 이제 RAID를 시작할 때 체계를 재기동하지 않고 그것을 올려태우기하며 /etc/rc.d/rc.local를 통하여 실행되게 하기 위하여 다음과 같이 한다. 즉

```
raidstart/dev/md0
mount/dev/md0/opt2
```

RAID1배치구성

RAID1준위를 설정하려면 etc/raidtab를 다음과 같이 편집한다.

```
raiddev              /dev/md0
   raid-level           1
   nr-raid-disks        2
   nr-spare-disks       0
   chunk-size           4
   persistent-superblock  1
   device              /dev/sda3
   raid-disk            0
   device             /dev/sda4
   raid/disk            1
```

실행결과를 관찰하면 모든것이 RAID−준위명령에 포함된다. 이제 다시 mkraid/dev/md0, mkfs −t ext2/dev/md0을 수행하고 raidstart/dev/md0로 RAID를 출발시키며 요구하는곳은 어디에나 이것을 올려태우기할수 있다. 속성예비디스크를 가진 RAID5는 자료의 안전성을 보장하는데서 상당히 쓸모 있는 객체라고 생각한다. 실행을 위한 모든 배치구성후 프로그람을 제외하고 알맞는 etc/raidtab를 아래에 보여 주었다.

```
raiddev           /dev/md0
    raid-level        5
    nr-raid-disks     7
    nr-spare-disks    1
    parity-algorithm  left-symmetric
    chunk-size       32
    device         /dev/sda1
    raid-disk         0
    device         dev/sdb1
    raid-disk        1
    device         dev/sdc3
    raid-disk        2
    device        dev/sdd4
    raid-disk        3
    device        dev/sde1
    raid-disk        4
    device        dev/sdf1
    raid-disk        5
    device        dev/sdg1
    raid-disk        6
    # here come the spare-disk
    device        /dev/sdh1
    spare-disk
```

RAID의 한계

RAID는 좋은것이 많은 반면에 일정한 제한성도 있다. 다시 강조하지만 어떤 종류의 쏘프트웨어 RAID에 체계등록부를 놓지 말아야 한다(특히/boot와 /usr).

기동시에 Linux는 여전히 RAID에 대하여 잘 알고 있지 못하며 따라서 제대로 초기화할수 없다. RAID는 오직 자료디스크로 쓸것만을 권고한다.

프로그람과 체계파일들은 RAID장치에 속하지 않는다. 왜냐하면 아주 중요한 /boot/sbin과 /usr등록부를 가지고 작업을 진행하려고 하면 좋지 않은 일이 생길수 있기때문이다. 사용자는 구획분할된 디스크상에서 거울조작도 실행할수 있으나 그것의 반대조작은 불가능하다. 따라서 여러개의 디스크상에 구획을 나누어 줄수 있으며 그것의 짝대기에 거울을 만들수 있다. 하지만 꼭 필요되는 일이라고 하더라도 거울자체를 구획분할할수 없다.

현재 가지고 있는 한개의 디스크로 RAID1의 절반을 거울로 설정할수 있으며 후에 새 디스크를 얻으면 거기로 거울을 내려 놓을수 있다. 하지만 한 디스크의 내용을 다른 디스크로 복사하는것과 그것들을 쌍으로 묶는것이 아주 어렵기때문에 그대로 할것을 권고하지는 않는다. 이 방법을 실현하기 위하여서는 우선 테프라든가 혹은 세번째 디스크의 자료를 여벌복사하여 그것을 두번째 거울디스크가 추가된 다음에 회복하여야 한다.

또 다른 방법은 dev/null로 두번째 RAID1을 정의하고 그다음 실제 디스크를 etc/raidtab의 두번째 디스크로써 설정하는 방법이다. 두개 디스크배치구성을 위한 RAID1과 RAID5사이의 차이점은 무엇인가?(즉 두개 디스크밖의 RAID1배렬과 두개 디스크밖의 RAID5배렬의 차이)

기억용량에서는 아무런 차이도 없으며 용량을 증가시키기 위하여서는 매개 배렬에 디스크들을 추가할수 있다(자세한 내용은 아래에서 본다.).

RAID1은 I/O읽기에서 우월한 성능을 제공한다. 즉 RAID1구동기는 2개 쎅터 즉 매 구동기로부터 각각 하나씩 동시적으로 읽을수 있는 분산-읽기기능을 리용하며 따라서 2중읽기성능을 발휘한다. RAID5구동기는 여러가지 최량적수법을 포함하지만 실제적으로 자료디스크의 거울복사로 되는 기우성디스크를 실현하지 못하고 있다. 따라서 여기서는 자료를 직렬로 읽는다.

RAID장치의 오유회복

일부 RAID알고리듬은 다중디스크오유에 대한 관측을 하고 있으나 현재 Linux에서는 실현하지 못하고 있다. 그러나 Linux 쏘프트웨어RAID는 배렬의 짝대기에 관하여 배렬을 계층화함으로써 다중디스크오유에 대하여 관측할수 있다. 실례로 9개의 디스크로는 3개의 RAID5배렬을 만들수 있다. 이 3개의 배렬은 짝대기로부터 단일 RAID5배렬을 호상 차례로 련결시켜 구성할수 있다. 사실 이런 형식의 배렬은 3개 디스크오유에 대하여 관측할수 있다. 큰 디스크공간은 여유정보에 관하여 《랑비》된다는데 대하여 강조한다.

동작이 중지되지 않은 상태에서 구획들은 다음상태들중의 어느 하나에 있을수 있다.

- ▼ 기억디스크의 캐쉬는 불확정중지가 발생할 때 설정되는 RAID로서 sync상태에 있다. 잃어 지는 자료는 없다.
- 기억디스크의 캐쉬는 폭주가 발생할 때 내용을 설정한 RAID보다 더 새롭게 갱신된다. 이것은 파일체계가 흐트러 지는 결과를 발생하며 잠재적인 자료들을 잃게 된다. 이 상태는 다음의 두개 상태로 더 나누어 진다.
  - Linux는 불확정중지가 발생할 때 자료를 쓴다.
  - Linux는 폭주가 발생할 때 자료를 쓰지 않는다.

이제 RAID1배렬을 리용한다고 하자. 우의 두 경우에 상태는 폭주되기전 상태일수도 있다. 이때 적은 수의 자료블로크들이 몇개의 거울들에만 성과적으로 씌여 지며 따라서 다음 재기동시에 거울에는 같은 자료가 더이상 포함되지 않는다. 만약 거울의 차이를 피하려고 했다면 RAID구동기의 균형읽기코드는 자료읽기를 임의의 거울로부터 선택하며 이로부터 거울들은 불일치동작을 야기시키게 된다. 만일 RAID구획이 명백히 올려태우기 해제되지 않으면 fsck가 실행되며 자체로 파일체계를 정돈한다. RAID구획을 정비하거나 혹은 회복하기 위한 ckraid−fix명령도 있다. 제일 좋기는 /etc/rc.local상에 이 명령을 주어 매번 체계가 기동할 때 이것을 실행시키는것이다. 이 조작은 /etc/rc.d/rc.local에 다음과 같은 명령행을 추가하는 방법으로 실현할수 있다.

```
mdadd /dev/md0 /dev/sda1 /dev/sdc1 ll{
ckraid --fix /etc/raid.usr.conf
mdadd/dev/md0 /dev/hda1 /dev/hdc1
}
```

혹은

```
mdrun -p1/dev/md0
if [$? -gt 0];then
ckraid --fix/etc/raid1.conf
mdrun  -p1/dev/md0
fi
```

기정값에 의하여 ckraid−fix는 첫번째 동작거울을 선택하며 그것의 내용으로 다른 거울들의 내용을 갱신한다. 그러나 폭주가 발생하는 정확한 시간에 따라 다른 거울의 자료들이 더 갱신될수 있으며 그것을 원천거울로 대신 리용해도 된다.

실례 1.

사용자에게 설정된(거울로 된) RAID1이 있고 디스크동작이 진행될 때 전원이 차단되였다.

해답. RAID의 준위여유도는 디스크의 고장을 막을수 있게 설계되였으나 전원고장에 대해서는 고려하지 못하였다. 이러한 상태에서 회복시킬수 있는 방법이 몇가지 있다. RAID배렬을 동기화하는데 필요한 RAID도구들을 리용한다. 이 도구들은 파일체계고장을 수리하지는 못한다. RAID배렬을 동기화시킨 다음 파일체계를 fsck에 의하여 수리해야 한다. RAID배렬은 ckraid/etc/raid.conf(RAID1에 대하여 만일 아니면 etc/raid5.conf 등을 리용한다.)로 검사할수 있다. ckrai/etc/raid1.conf−fix을 호출하여 배렬안의(보통 첫번째) 디스크들중에서 하나를 끄집어 내여 기본복사(master copy)로 리용하며 그 블로크를 거울의 다른 블로크에 복사한다. 디스크들중의 어느것이 기본디스크로 사용되여야 하는것을 지적하기 위하여 −force −source기발을 리용할수 있다. 실례로 ckraid/etc/raid1.conf −fix −force −source/dev/hdc3을 리용할수 있다. ckraid명령은 아무런 변경도 주지 않고 비능동 RAID배렬을 검증할수 있도록 −fix선택항목이 없이 안전하게 실행할수 있다. 제기된 변경을 시켜도 일 없을 때는 −fix선택항목을 준다.

실례 2.

사용자에게 설정된 RAID4 혹은 RAID5(기우성)가 있고 디스크동작이 진행될 때 전원이 차단되였다.

해답. RAID준위의 여유도는 디스크고장을 막을수 있게 설계되였으나 전원고장에 대해서는 그렇게 되지 않는다. RAID4나 RAID5배렬의 디스크들이 fsck가 읽을수 있는 파일체계를 포함하고 있지 않기때문에 우의 항목이 보다 더 적다. 사용자는 예비검사나 수리를 위하여 fsck를 사용할수 없으며 반드시 ckraid를 우선 리용해야 한다. ckraid명령은 어떤 명령도 주지 않고 비능동상태의 RAID배렬을 검증할수 있게 −fix선택항목을 주지 않고 안전하게 실행시킬수 있다. 제기된 변경을 주어도 일 없을 때는 −fix선택항목을 준다. 만일 요구되면 "failed disk"(고장난 디스크)로써 디스크들중의 어느 하나를 지적할수 있다. 이 조작을 위하여서는 −suggest −failed −disk –mask flag를 수행한다. 기발안에서 한 비트만이 설정되여야 한다. RAID5는 고장난 디스크 두개는 회복시키지 못한다.

여기서 마스크는 2진수비트마스크이며 다음과 같다.

```
0x1==first disk
0x2==second disk
0x4==third disk
0x8==fourth disk, etc.
```

한편 사용자는 suggest −fix −parity기발을 리용하여 기우성섹터를 교정하도록 선택할수 있다. 이 조작에서는 다른 섹터들로부터 기우성을 다시 계산한다.

flags −suggest −faild −dsk −mask와 suggest −fix −parity는 검증에 리용될수 있다.
−fix기발이 정의되지 않으면 아무러한 변경도 이루어 지지 않는다. 따라서 우리는 서로 다른 가능한 수리도식들로 실현해 볼수 있다.

```
/*
   md_k.h : kernel internal structure of the Linux MD driver
          Copyright (C) 1996-98 Ingo Molnar, Gadi Oxman

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   You should have received a copy of the GNU General Public License
   (for example /usr/src/linux/COPYING); if not, write to the Free
   Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

#ifndef _MD_K_H
#define _MD_K_H

#include <linux/kernel.h>    // for panic()

#define MD_RESERVED       0UL
#define LINEAR            1UL
#define RAID0             2UL
#define RAID1             3UL
#define RAID5             4UL
#define TRANSLUCENT       5UL
#define HSM               6UL
#define MULTIPATH         7UL
#define MAX_PERSONALITY   8UL

extern inline int pers_to_level (int pers)
{
      switch (pers) {
            case MULTIPATH:   return -4;
            case HSM:         return -3;
            case TRANSLUCENT:return -2;
```

```
		case LINEAR:	return -1;
		case RAID0:	return  0;
		case RAID1:	return  1;
		case RAID5:	return  5;
	}
	panic("pers_to_level()");
	return 0;
}

extern inline int level_to_pers (int level)
{
	switch (level) {
		case -4: return MULTIPATH;
		case -3: return HSM;
		case -2: return TRANSLUCENT;
		case -1: return LINEAR;
		case  0: return RAID0;
		case  1: return RAID1;
		case  4:
		case  5: return RAID5;
	}
	return MD_RESERVED;
}

typedef struct mddev_s mddev_t;
typedef struct mdk_rdev_s mdk_rdev_t;

#if (MINORBITS != 8)
#error MD doesnt handle bigger kdev yet
#endif

#define MAX_MD_DEVS  (1<<MINORBITS)	/* Max number of md dev */

/*
 * Maps a kdev to an mddev/subdev. How 'data' is handled is up to
 * the personality. (eg. HSM uses this to identify individual LVs)
 */
typedef struct dev_mapping_s {
	mddev_t *mddev;
```

```
	void *data;
} dev_mapping_t;

extern dev_mapping_t mddev_map [MAX_MD_DEVS];
extern inline mddev_t * kdev_to_mddev (kdev_t dev)
{
	if (MAJOR(dev) != MD_MAJOR)
		BUG();
	 return mddev_map[MINOR(dev)].mddev;
}

/*
 * options passed in raidrun:
 */

#define MAX_CHUNK_SIZE (4096*1024)

/*
 * default readahead
 */
#define MD_READAHEAD	MAX_READAHEAD

extern inline int disk_faulty(mdp_disk_t * d)
{
	return d->state & (1 << MD_DISK_FAULTY);
}

extern inline int disk_active(mdp_disk_t * d)
{
	return d->state & (1 << MD_DISK_ACTIVE);
}

extern inline int disk_sync(mdp_disk_t * d)
{
	return d->state & (1 << MD_DISK_SYNC);
}

extern inline int disk_spare(mdp_disk_t * d)
{
```

```
        return !disk_sync(d) && !disk_active(d) && !disk_faulty(d);
}

extern inline int disk_removed(mdp_disk_t * d)
{
        return d->state & (1 << MD_DISK_REMOVED);
}

extern inline void mark_disk_faulty(mdp_disk_t * d)
{
        d->state |= (1 << MD_DISK_FAULTY);
}

extern inline void mark_disk_active(mdp_disk_t * d)
{
        d->state |= (1 << MD_DISK_ACTIVE);
}

extern inline void mark_disk_sync(mdp_disk_t * d)
{
        d->state |= (1 << MD_DISK_SYNC);
}

extern inline void mark_disk_spare(mdp_disk_t * d)
{
        d->state = 0;
}

extern inline void mark_disk_removed(mdp_disk_t * d)
{
        d->state = (1 << MD_DISK_FAULTY) | (1 << MD_DISK_REMOVED);
}

extern inline void mark_disk_inactive(mdp_disk_t * d)
{
        d->state &= ~(1 << MD_DISK_ACTIVE);
}

extern inline void mark_disk_nonsync(mdp_disk_t * d)
```

```
{
      d->state &= ~(1 << MD_DISK_SYNC);
}

/*
 * MD's 'extended' device
 */
struct mdk_rdev_s
{
      struct md_list_head same_set;/* RAID devices within the same set */
      struct md_list_head all;     /* all RAID devices */
      struct md_list_head pending; /* undetected RAID devices */

      kdev_t dev;                  /* Device number */
      kdev_t old_dev;              /*  "" when it was last imported */
      unsigned long size;          /* Device size (in blocks) */
      mddev_t *mddev;              /* RAID array if running */
      unsigned long last_events;   /* IO event timestamp */

      struct block_device *bdev;   /* block device handle */

      mdp_super_t *sb;
      unsigned long sb_offset;

      int alias_device;            /* device alias to the same disk */
      int faulty;                  /* if faulty do not issue IO requests */
      int desc_nr;                 /* descriptor index in the superblock
*/
};

/*
 * disk operations in a working array:
 */
#define DISKOP_SPARE_INACTIVE   0
#define DISKOP_SPARE_WRITE      1
#define DISKOP_SPARE_ACTIVE     2
#define DISKOP_HOT_REMOVE_DISK  3
#define DISKOP_HOT_ADD_DISK     4
typedef struct mdk_personality_s mdk_personality_t;
```

```
struct mddev_s
{
	void			*private;
	mdk_personality_t	*pers;
	int			__minor;
	mdp_super_t		*sb;
	int			nb_dev;
	struct md_list_head		disks;
	int			sb_dirty;
	mdu_param_t		param;
	int			ro;
	unsigned int			curr_resync;
	unsigned long			resync_start;
	char			*name;
	int			recovery_running;
	struct semaphore	reconfig_sem;
	struct semaphore	recovery_sem;
	struct semaphore	resync_sem;
	struct md_list_head		all_mddevs;
	request_queue_t			queue;
};

	struct mdk_personality_s
{
	char *name;

	int (*map)(mddev_t *mddev, kdev_t dev, kdev_t *rdev),
		unsigned long *rsector, unsigned long size);
	int (*make_request)(mddev_t *mddev, int rw, struct buffer_head *
bh);
	void(*end_request)(struct buffer_head * bh, int uptodate);
	int (*run)(mddev_t *mddev);
	int (*stop)(mddev_t *mddev);
	int (*status)(char *page, mddev_t *mddev);
	int (*error_handler)(mddev_t *mddev, kdev_t dev);

/*
 * Some personalities (RAID-1, RAID-5) can have disks hot-added and
```

```
 * hot-removed. Hot removal is different from failure. (failure marks
 * a disk inactive, but the disk is still part of the array) The interface
 * to such operations is the 'pers->diskop()' function, can be NULL.
 *
 * the diskop function can change the pointer pointing to the incoming
 * descriptor, but must do so very carefully. (currently only
 * SPARE_ACTIVE expects such a change)
 */
	int (*diskop) (mddev_t *mddev, mdp_disk_t **descriptor, int state);

	int (*stop_resync)(mddev_t *mddev);
	int (*restart_resync)(mddev_t *mddev);
	int (*sync_request)(mddev_t *mddev, unsigned long block_nr);
};


/*
 * Currently we index md_array directly, based on the minor
 * number. This will have to change to dynamic allocation
 * once we start supporting partitioning of md devices.
 */
extern inline int mdidx (mddev_t * mddev)
{
	return mddev->__minor;
}

extern inline kdev_t mddev_to_kdev(mddev_t * mddev)
{
	return MKDEV(MD_MAJOR, mdidx(mddev));
}

extern mdk_rdev_t * find_rdev(mddev_t * mddev, kdev_t dev);
extern mdk_rdev_t * find_rdev_nr(mddev_t *mddev, int nr);
extern mdp_disk_t *get_spare(mddev_t *mddev);

/*
 * iterates through some rdev ringlist. It's safe to remove the
 * current 'rdev'. Dont touch 'tmp' though.
 */
```

```
#define ITERATE_RDEV_GENERIC(head,field,rdev,tmp)        \
                                                          \
      for (tmp = head.next;                               \
            rdev = md_list_entry(tmp, mdk_rdev_t, field), \
                  tmp = tmp->next, tmp->prev != &head     \
            ; )
/*
 * iterates through the 'same array disks' ringlist
 */
#define ITERATE_RDEV(mddev,rdev,tmp)                      \
      ITERATE_RDEV_GENERIC((mddev)->disks,same_set,rdev,tmp)

/*
 * Same as above, but assumes that the device has rdev->desc_nr numbered
 * from 0 to mddev->nb_dev, and iterates through rdevs in ascending order.
 */
#define ITERATE_RDEV_ORDERED(mddev,rdev,i)                \
      for (i = 0; rdev = find_rdev_nr(mddev, i), i < mddev->nb_dev; i++)

/*
 * Iterates through all 'RAID managed disks'
 */
#define ITERATE_RDEV_ALL(rdev,tmp)                        \
      ITERATE_RDEV_GENERIC(all_raid_disks,all,rdev,tmp)

/*
 * Iterates through 'pending RAID disks'
 */
#define ITERATE_RDEV_PENDING(rdev,tmp)                    \
      ITERATE_RDEV_GENERIC(pending_raid_disks,pending,rdev,tmp)

/*
 * iterates through all used mddevs in the system.
 */
#define ITERATE_MDDEV(mddev,tmp)                          \
                                                          \
      for (tmp = all_mddevs.next;                         \
            mddev = md_list_entry(tmp, mddev_t, all_mddevs),    \
                  tmp = tmp->next, tmp->prev != &all_mddevs     \
```

```
            ; )

extern inline int lock_mddev (mddev_t * mddev)
{
      return down_interruptible(&mddev->reconfig_sem);
}

extern inline void unlock_mddev (mddev_t * mddev)
{
      up(&mddev->reconfig_sem);
}

#define xchg_values(x,y) do { __typeof__(x) __tmp = x; \
                        x = y; y = __tmp; } while (0)

typedef struct mdk_thread_s {
      void              (*run) (void *data);
      void              *data;
      md_wait_queue_head_t   wqueue;
      unsigned long          flags;
      struct semaphore *sem;
      struct task_struct     *tsk;
      const char        *name;
} mdk_thread_t;

#define THREAD_WAKEUP  0

#define MAX_DISKNAME_LEN 64

typedef struct dev_name_s {
      struct md_list_head list;
      kdev_t dev;
      char namebuf [MAX_DISKNAME_LEN];
      char *name;
} dev_name_t;

#define __wait_event_lock_irq(wq, condition, lock)  \
do {                                                \
      wait_queue_t __wait;                          \
```

```
	init_waitqueue_entry(&__wait, current);      \
	                                             \
	add_wait_queue(&wq, &__wait);                \
	for (;;) {                                   \
		set_current_state(TASK_UNINTERRUPTIBLE);\
		if (condition)                          \
			break;                          \
		spin_unlock_irq(&lock);                 \
		run_task_queue(&tq_disk);               \
		schedule();                             \
		spin_lock_irq(&lock);                   \
	}                                            \
	current->state = TASK_RUNNING;               \
	remove_wait_queue(&wq, &__wait);             \
} while (0)

#define wait_event_lock_irq(wq, condition, lock)    \
do {                                                \
	if (condition)                              \
		break;                              \
	__wait_event_lock_irq(wq, condition, lock); \
} while (
#endif
```

제7장. 2차확장파일체계

이 장에서는 Linux파일체계에서 가장 널리 리용되는 2차확장파일체계(Second Extended File System, ext2fs)에 대하여 구체적으로 고찰한다.

확장파일체계의 기본개정판은 Remy Card, Theodore Ts'o, Stephen Twedie에 의하여 서술되였으며 1993년 1월에 처음으로 발표되였다. 현재 이 체계는 Linux에 의하여 리용되고 있는 유력한 파일체계이다.

이외에 NetBSD, FreeBSD, GNU HURD와 Windows95/98/NT, OS/2 그리고 RISC OS에 리용할수 있는 실현판들도 있다.

ext2fs는 1차 확장파일체계(first Extended File System)에서 제기된 문제점을 수정하고 강력한 파일체계를 제공하기 위하여 설계되고 실현되였다. 이 파일체계는 UNIX의 의미론적방법을 실현하고 있으며 보다 고급한 특성들을 제공하고 있다.

새로운 특성

물론 설계자들도 역시 뛰여 난 성능을 얻기 위하여 ext2fs를 요구하였으며 콤퓨터를 집중적으로 리용하는데서 자료가 루실될 위험성을 줄이기 위해서도 적응성이 좋은 파일체계를 요구하였다.

마지막에 언급하겠지만 ext2fs가 사용자로 하여금 파일체계의 재양식화를 하지 않고도 새로운 특성으로 하여 리득을 얻을수 있도록 확장가능한 항목들을 포함해야 하였다.

표준ext2fs의 특성

ext2fs는 표준 UNIX파일형식들 즉 정규파일, 등록부, 장치전용파일 그리고 기호적련결들을 지원한다. Ext2fs는 실제적으로 큰 구획에 생성된 파일체계들을 관리할수 있다. 초기의 핵심부코드는 파일체계의 최대크기가 2GB로 제한되여 있었으나 VFS층에서 동작하는 현재 파일체계는 4TB로 늘어 났다.

따라서 지금은 구획을 많이 만들지 않고도 큰 디스크들을 사용할수 있다. ext2fs는 255개 문자까지 긴 파일이름을 쓸수 있는 능력을 제공하고 있으며 여러가지 길이의 등록부입구점들을 사용할수 있다. 파일이름의 한계는 필요한 경우 1012까지도 확장할수 있다. ext2fs는 상위블로크(root)를 위하여 블로크의 5%까지를 예약하고 있기때문에 관리기로 하여금 사용자프로쎄스들이 파일체계를 완전히 써버리는 경우에도 쉽게 회복할수 있다.

개선된 ext2fs의 특성

표준UNIX의 특성외에 ext2fs는 일반적인 UNIX파일체계에 주어 져 있지 않는 많은 확장성을 지원한다. 파일속성들은 핵심부의 동작이 파일모임우에서 진행될 때 사용자가

그것을 변경시킬수 있게 해준다.

사용자는 파일이나 등록부상에 속성을 설정할수 있다. 등록부에 속성을 설정하는 경우에 그 등록부에 생성되는 새 파일들은 이 속성들을 계승해야 한다. BSD 혹은 System V Release4의 의미론은 올려태우기시에 설정된다.

올려태우기의 선택은 관리기가 파일생성의미론을 선택할수 있게 한다. BSD의 의미론으로 올려태우기된 파일체계상에서 파일은 그것의 선포등록부와 같은 그룹 id(식별자)로 생성된다. System V의미론은 좀더 복잡하다. 만일 등록부의 setgid비트가 설정되여 있으면 새 파일은 그 등록부의 그룹 id를 계승한다. 한편 부분등록부들은 그룹 id와 setgid비트를 계승한다. 한편 파일과 부분등록부들은 호출프로쎄스의 원시그룹 id를 가지고 생성된다.

BSD류형의 동기갱신조작들이 ext2fs에서 리용될수 있다. 올려태우기선택은 관리기로 하여금 메타자료(색인마디, 비트매프블로크, 간접블로크, 등록부블로크)를 그것들이 변경될 때 디스크상에 동기적으로 쓸것을 요청하게 한다.

이것은 엄밀한 메타자료의 일관성을 유지하는데 쓸모가 있지만 체계의 성능을 약화시키는 결과를 초래할수 있다. 현실에서 이 특성은 일반적으로 리용되지 않으며 게다가 메타자료의 동기적갱신의 리용과 관련한 성능이 더 떨어 지기 때문에 사용자자료 즉 파일체계검사기에 의하여 기발에 표시되지 않는 자료에 헝클어짐이 발생할수 있다.

ext2fs는 또한 관리기들이 파일체계를 만들 때 론리적블로크의 크기도 선택할수 있게 한다. 블로크크기는 전형적으로 1024, 2048, 4096byte로 될수 있다. 큰 블로크크기를 리용하면 파일에 접근하는데 필요한 I/O요청수가 더 적고 또 디스크머리부찾기수도 더 적어지기때문에 속도를 높일수 있다.

달리 말하면 큰 블로크들은 더 많은 기억공간을 절약할수 있게 하며 평균적으로 파일에 배정된 마지막블로크는 절반이면 된다. 따라서 블로크들을 더 크게 취하면 더 많은 공간이 절약된다.

또한 보다 큰 블로크크기를 리용하는데서 주되는 우점은 ext2fs파일체계의 선행배정기술에 의하여 얻어 진다.

마지막으로 ext2fs는 기호련결을 고속화한다. 고속기호련결은 파일체계상의 임의의 자료블로크들을 리용하지 못한다. 목적이름이 자료블로크에는 기억되지 않고 색인마디자체에 기억된다. 이 방법은 배정되여야 할 자료블로크가 없기때문에 일정한 디스크공간을 기억할수 있으며 련결에 의하여 접근할 때에는 자료블로크의 읽기가 필요없으므로 련결연산의 속도를 높인다.

물론 색인마디에서 리용할수 있는 공간은 제한되며 따라서 매개 련결은 고속기호련결로 실현될수 있다. 고속기호련결에서 목표이름의 최대길이는 60문자이다. 가까운 앞날에 파일들을 작게 할수 있도록 이 도식을 확장하기 위한 연구가 진행되고 있다. 기호련결들은 또한 색인마디를 가지는 파일체계객체들이다. 련결들은 symlink련결이 60byte보다 작으면 그것들을 위한 자료가 색인마디자체안에 기억되기때문에 특별히 강조하여야 한다. 색인마디는 자료기억을 위한 블로크에 대한 지적자를 보존하는데 표준적으로 리용될수 있는 마당들을 가지고 있다. 이것은 련결이 블로크를 대상하지 않도록 하기 위한 하나의 최량화이다.

등 록 부

등록부는 파일체계의 객체이며 파일과 같은 색인마디를 가지고 있다. 등록부는 색인마디의 번호와 함께 매개 이름과 관련된 레코드를 포함하고 있는 특별히 양식화된 파일이다. 파일체계의 이후 개정판들도 역시 객체의 형 즉 파일, 등록부, 기호련결장치, 대기렬 그리고 속도를 제고할 목적으로 등록부안의 소케트를 부호화하고 있다. ext2의 실현판은 등록부안에 련결목록을 리용하고 있으며 계획적으로 성능을 높이기 위하여 B나무를 대신 리용한다. 또 현재의 실현에서는 보다 큰 파일을 수용하기 위하여 등록부가 커지기만 하면 그 등록부들을 절대로 줄이지 않는다.

문자장치나 블로크전용장치들은 등록부로 지정되는 자료블로크를 포함할수 없다. 그대신 블로크를 지적하는데 리용될 마당들을 다시 재리용하여 색인마디에 장치번호를 기억시킨다. ext2fs는 파일체계의 상태를 기억하고 있다. 파일체계의 상태를 지적하기 위하여 핵심부코드가 상위블로크의 특정한 마당을 리용한다. 파일체계가 읽기/쓰기방식으로 올려태우기될 때 그것의 상태는 “Not Clean”으로 설정된다. 또 읽기방식에서 올려태우기해제되거나 재올려태우기될 때에는 상태가 “clean”으로 재설정된다.

기동시 파일체계검사기는 파일체계를 검사해야 되겠는가 안해도 되겠는가를 결정하기 위하여 이 정보를 리용한다. 핵심부코드도 역시 이 마당에 오유를 기록한다. 핵심부코드에 의하여 불일치성이 검출될 때 파일체계는 “Erroneons”로 표기된다.

파일체계검사기는 상태가 외관상 명백하게 보여도 상관없이 파일체계의 검사를 강하게 요구하기 위하여 이 마당을 검열한다. 때로는 파일체계검사를 건너 뛰는 경우가 있는데 이 조작은 위험할수 있기때문에 ext2fs는 정기적인 간격으로 검사를 진행한다.

상위블로크에는 올려태우기계수기가 보존되여 있다. 파일체계가 읽기/쓰기방식으로 올려태우기될 때마다 이 계수기가 증가된다. 이 계수기값이 최대값에 도달하면(상위블로크에 기록된) 파일체계검사기는 파일체계가 “Clran”상태에 있다 하더라도 검사를 진행한다. 마지막 검사시간과 최대검사간격도 역시 상위블로크에 기억된다.

이 두 마당은 관리기가 주기적인 검사를 요청할수 있게 한다. 최대검사간격에 도달되면 검사기는 파일체계상태를 무시하고 파일체계의 검사를 실행시킨다. ext2fs는 또한 파일체계의 동작을 조정하기 위한 도구도 제공한다. Tune2fs프로그람은 다음과 같은것들을 변경시키는데 리용된다.

▼ 오유동작. 핵심부코드에 의하여 불일치성이 검출되면 파일체계는 “Erroneous”로 표식되며 다음의 3가지 동작중에서 어느 하나를 취한다. 즉 련속적인 정상실행, 파일체계의 파괴를 피하기 위한 읽기전용방식으로의 파일체계재올려태우기, 파일체계검사를 실행하기 위한 재기동
- 최대올려태우기계수기
- 최대검사간격

▲ 상위블로크에 예약된 론리적블로크수

올려태우기선택(option)핵심부오유동작을 변경시키는데도 리용될수 있다. 속성은 사용

자가 파일에 관하여 안전한 지우기를 요청하는데 리용된다. 파일이 지워 지면 우연자료(randomdata)가 파일에 이미 배정된 디스크블로크에 기록된다. 이것은 해커들과 디스크편집기를 리용하여 이미 있던 파일의 내용에 접근할 음흉한 목적을 가진 사람들로부터 체계를 보호하는데 리용된다. 마지막으로 4.4BSD파일체계에 의하여 시사된 새형의 파일체계들이 현재 ext2fs에 보충되고 있다는것을 강조해 둔다.

불변파일들에 대하여서는 읽기만이 가능하며 쓰거나 지우기는 할수 없다. 이 파일들은 예민한 배치구성파일들을 보호하는데 리용될수 있다.

추가전용파일들은 쓰기방식으로 열수 있는데 자료는 항상 파일의 끝에 첨부된다. 불변파일들과 같이 이 파일들은 지울수 없고 이름을 재정의할수 없다. 이 파일은 특히 크기가 커지기만 하는 가동일지파일들에 쓸모가 있다.

블로크

디스크나 파일의 공간은 블로크로 나누어 진다. 이 블로크들의 크기는 1024, 2048, 4096byte로 고정되여 있다. 블로크의 크기는 파일체계가 생성될 때 결정된다. 보다 작은 블로크일수록 파일당 공간랑비가 더 작아 진다는것을 알수 있지만 대신 휴지시간이 더 길어 지게 된다. 블로크들은 단편화되는것과 다량의 련속자료를 읽을 때 머리부찾기회수를 줄이기 위하여 블로크그룹으로 클라스터화(cluster)하여 리용한다. 매 블로크그룹은 서술자와 상위블로크뒤에 직접 기억되는 서술자배렬을 가지고 있다.

매 그룹선두의 두개 블로크는 블로크전용비트매프와 어느 블로크와 색인마디가 리용되는가를 보여 주는 색인마디전용비트매프용으로 예약되여 있다. 매개 비트매프는 블로크와 일치되기때문에 블로크그룹의 최대크기는 블로크크기의 8배라는것을 알수 있다. 블로크그룹의 첫 블로크(예약되지 않는)는 블로크용의 색인마디표를 가리키며 나머지는 자료블로크들이다.

블로크배정알고리듬은 색인마디가 블로크그룹들을 포함하고 있다는데로부터 같은 블로크내에 자료블로크들을 배정하게 한다.

예약공간

ext2는 특정한 사용자(표준으로는 super-user)에 대하여 일정한 블로크수를 예약할수 있는 기능을 가지고 있는데 이 특정한 사용자는 체계로 하여금 어떤 사용자가 리용가능한 모든 공간을 다 채웠다고 해도 동작을 계속 할수 있게 해준다. 또한 완전히 공간을 다 채운후에도 파일체계를 계속 보존하며 이것은 공간이 단편화되는것을 막을수 있게 한다.

파일체계검사

기동시에 대다수 체계들은 파일체계에 대한 일관성검사(e2fsck)를 진행한다. ext2파일체계의 상위블로크는 기동시에 파일체계가 너무 크면 검사하는데 시간이 오래 걸리기때문에 실지 fsck가 실행되는지 안되는지를 지적하는 몇개의 마당을 가지고 있다. fsck는 파일체계가 오유없이 올려태우기되거나 혹은 최대올려태우기계수가 진행되고 있거나 혹은 검사들사이의 최대시간이 계측되고 있을 때 실행된다.

색인마디

색인마디(첨수마디)는 ext2파일체계에서 기본개념이다. 파일체계에서 매 객체는 색인마디에 의하여 표현된다. 색인마디구조체는 객체안에 보존된 자료와 그것의 이름을 제외한 객체에 관한 모든 메타자료를 포함하는 파일체계블로크들에 대한 지적자를 포함하고 있다.

메타자료는 허가권, 소유자, 그룹, 기발, 범위, 사용된 블로크수, 접근시간, 변경시간, 수정시간, 지운시간, 련결수, 단편수, 판본(NFS에서) 그리고 ACL들로 구성된다.

색인마디구조체에는 현재 리용되지 않는 몇개의 예약마당과 중복적재되는 예약마당들이 존재한다. 현재 마당은 색인마디가 한개 등록부이면 등록부 ACL에 그리고 색인마디가 정규파일이면 파일크기의 웃 32bit에 리용된다.

변환마당은 Linux에서는 리용되지 않지만 HURD에서 이 객체를 해석하기 위하여 리용될 프로그람의 색인마디를 참조하는데 사용된다. HURD는 또한 보다 큰 허가권, 소유자, 그룹마당을 가지며 따라서 여기서는 여유비트들을 기억하기 위하여 쓰이지 않는 몇개의 다른 마당을 리용한다.

색인마디안의 파일자료를 포함하는 첫 12개 블로크에 대한 지적자들이 있다. 또한 다음블로크모임을 지적하는 지적자들을 포함하는 간접블로크지적자와 간접블로크에 대한 지적자를 포함하는 2중간접블로크지적자 그리고 2중간접블로크지적자를 포함하는 3중간접블로크지적자가 있다.

기발마당은 표준chmod기발들에 의하여 제공되지 않는 몇개의 ext2−전용기발들을 포함한다. 이 기발들은 lsattr로 목록화될수 있으며 chattr명령으로 변경시킬수 있다. 여기에는 또한 지우기, 지우기불가능, 압축, 동기화갱신, 불변성, 추가전용, dumptable, no−atime, B나무등록부기능을 안전하게 실행할수 있는 기발들이 포함된다.

상위블로크

상위블로크는 파일체계의 배치구성에 대한 모든 정보를 포함하고 있다. 상위블로크는 파일체계의 블로크1(0으로부터 번호를 매긴)에 기억되며 올려태우기하는데서 본질적요소로 된다. 상위블로크가 아주 중요하기때문에 상위블로크의 여벌복사는 파일체계전반에 걸쳐 블로크그룹들에 기억된다.

ext2의 첫번째 개정판은 매개 블로크그룹의 시작에 복사판을 보관한다. 두번째 개정판은 대규모파일체계상에서 여유도를 줄일수 있도록 같은 블로크그룹에만 복사판을 보존한 정의된 그룹은 0, 1과 3, 5, 7의 제곱으로 된다.

상위블로크는 파일체계에 색인마디와 블로크가 몇개 있는가, 그중에서 몇개가 리용되지 않고 있는가, 블로크그룹안에 몇개의 색인마디와 블로크가 있는가 그리고 언제 파일체계가 올려태우기되였으며 언제 수정되였는가, 파일체계의 판본은 얼마이고 어느 OS가 생성하였는가와 같은 정보를 포함하고 있다.

파일체계가 최근에 개정되였으면 거기에는 기록권이름유일식별자, 색인마디크기, 압축지원, 블로크와 선행배정 그리고 상위블로크의 여벌복사를 적게 할 필요성 등이 포함된다. 상위블로크의 모든 마당(다른 ext2fs구조체에서와 같이) 들은 little endian양식으로

디스크에 기억되며 따라서 파일체계는 기계들사이에서 서로 이식될수 있으며 어느 기계가 그것을 생성했는가를 알 필요가 없다.

개 정

ext2에 리용된 개정기술은 아주 정교하다.

ext2의 판본 0(EXT2−GOOD−OLD−REV)에 의하여서는 지원되지 않지만 판본 1에는 도입되였다. 여기에는 3개의 32bit마당이 있는데 하나는 호환특성을 위한것이고 다른 하나는 읽기전용호환특성을 위한것이며 또 다른 하나는 비호환특성을 위한것이다.

물리적구조

ext2의 물리적구조에 강한 영향을 주는것은 BSD파일체계의 형식이다. ext2파일체계는 블로크그룹들로 구성되였으며 이 그룹들은 BSDFFS의 실린더그룹과 류사하다. 하지만 현대적장치들이 순차접근을 최량화하고 있으며 그것의 물리적공간형식을 조작체계로부터 감추려고 하고 있기때문에 블로크그룹들은 디스크상의 블로크들의 물리적륜곽에 구애되지 않는다.

파일체계의 물리적구조는 다음과 같다.

BOOT sector	BLOCK Group1	BLOCK Group2	…	BLOCK GroupN

매 블로크그룹은 아주 중요한 파일체계조종정보(상위블로크와 파일체계서술자)에 대한 예비복사판을 포함하고 있으며 또 파일체계의 다른 부분(블로크비트매프, 색인마디비트매프, 색인마디표의 한 부분, 자료블로크)도 포함한다.

블로크그룹의 구조는 다음과 같다.

Super Block	FS Description	Block Bitmap	Inode Bitmap	Inode Table	Data Block

블로크그룹의 리용은 현실적으로 아주 큰 우월성을 가진다. 조종구조가 매개 블로크그룹안에 복제되기때문에 상위블로크가 손상될 때 파일체계로부터 쉽게 복구할수 있으며 이 구조는 또한 성능도 개선할수 있게 한다. 색인마디표와 자료블로크들사이의 거리를 줄임으로써 파일 I/O수행시 디스크선두탐색시간을 줄일수 있게 한다. ext2fs에서 등록부들은 서로 다른 길이를 가진 입구점들의 련결목록으로서 관리된다. 매개 입구점은 색인마디번호, 입구점길이, 파일이름, 파일이름길이를 포함한다. 가변길이입구점을 리용하여 등록부안에서 기억공간을 랑비하지 않고 긴 파일이름도 처리할수 있다.

등록부입구점의 구조는 다음과 같다.

```
Inode number entrylength namelength filename
```

례를 들어 이 표가 세개의 파일 즉 file, long_file_name, f2을 가지는 등록부의 구조를 표현한다면 다음과 같이 된다.

i1	16	05	File1
i2	40	14	Long −file −name
i3	12	02	f2

성능최량화

ext2fs핵심부코드는 성능최량화수법이 포함되여 있으며 이것은 파일의 읽기나 쓰기시에 I/O의 속도를 높인다.

ext2fs는 선행읽기의 실행에 의하여 캐쉬관리에서 우월성이 나타나고 있다. 블로크를 읽어야 할 경우 핵심부코드는 여러개의 린접한 블로크들에 대하여 I/O를 요청한다. 이것은 읽어야 할 다음 블로크가 이미 캐쉬에 적재되였는가를 확인하려는데 목적이 있다는것을 의미한다. 선행읽기는 보통 파일에 대한 순차읽기를 진행할 때 수행되며 ext2fs에서는 이것을 명백한 읽기(readdir(2))나 혹은 암시적읽기(name:핵심부등록기검색)와 같은 등록기의 읽기로 확장하였다.

ext2fs는 또한 여러가지 배정최량화도 실현하고 있으며 블로크그룹들은 색인마디와 자료와 관련되는 클라스터에 리용되며 핵심부코드는 항상 동일한 그룹의 파일에 관한 자료블로크들을 색인마디로 배정하게 한다. 파일에 대하여 자료를 쓸 때 ext2fs는 린접한 블로크를 8개까지 선행배정한다. 선행배정적중률은 완전히 채워 진 파일체계상에서도 거의 75%에 도달한다.

이 선행배정방식은 적재량이 많은 조건에서 높은 성능을 가진다. 또한 이 방법은 련속된 블로크들을 파일로 배정할수 있게 하며 따라서 순차읽기의 속도는 더 높아 지게 된다.

이 두가지 배정최량화에 대하여 소개한다.

▼ 관계되는 파일로부터 블로크그룹
▲ 관계되는 블로크로부터 블로크배정을 위한 8bit 무리짓기

메타-자료(meta-data)

ext2에서 비동기적메타자료의 쓰기방식이 FFS동기적메타자료도식보다 더 빠르지만 실현성이 적다는데 대하여서는 흔히 론의되군 한다. 이 두 방법은 각각의 fsck프로그람으로 동일하게 해결될수 있다.

메타자료의 쓰기를 동기화하기 위한 방법에는 3가지 방법이 있다.

▼ 원천을 가지고 있으면 매 파일당 처리 : open()함수에 대하여 O_SYNC인수를 리용
• 원천을 가지고 있지 않을 때 매 파일당 처리 : chattr+s리용
▲ 파일체계당 처리 : mount−o sync

첫번째 방법과 세번째 방법은 ext2에서는 정의되지 않지만 메타자료를 동기적으로 쓸수 있다.

ext2fs서고

ext2fs서고는 사용자방식프로그람이 ext2파일체계의 조종구조체를 조종할수 있게 개발되였다. 이 서고는 물리적장치를 통하여 파일체계에 직접 접근함으로써 ext2파일체계상의 자료를 검열하거나 수정하는데 리용할수 있는 부분프로그람들을 제공한다.

ext2fs서고는 최대코드를 프로그람추상화기술을 리용하여 재리용할수 있게 한다. 실례로 몇개의 서로 다른 반복기가 제공된다. 프로그람은 색인마디안의 매 블로크를 호출하는 ext2fs_block−interate()에 한개의 기능으로 쉽게 넘길수 있다. 다른 반복기함수는 사용자제공함수가 등록부안의 매개 파일을 호출할수 있게 한다.

많은 ext2fs 편의프로그람 (Mke2fs, e2fsck, tune2fs, dump2fs, debugfs) 들은 ext2fs서고를 리용한다. 이것은 ext2파일체계방식에서의 새로운 특성들을 반영하기 위한 여러가지 변화들이 한가지 장소 즉 ext2fs서고에서만 만들어 져야 하기때문에 편의프로그람들의 유지보수성이 매우 단순해 진다. 이 코드들의 재리용은 ext2fs서고가 공유서고이메지로 만들어 진다는데로부터 2진파일의 크기가 보다 작아 지게 한다.

ext2fs서고의 대면부가 아주 추상화되고 또 일반적이므로 ext2fs파일체계에 직접 접근을 요구하는 새 프로그람은 읽기가 아주 쉽다.

실례로 ext2fs서고는 4.4BSD를 포구에 dump할 때 리용되며 또 편의프로그람들을 복구할 때 리용된다. 이러한 도구들을 Linux에 적응시키는데는 변경시킬 내용이 거의 없으며 극히 적은 체계의존함수들만이 ext2fs서고의 호출에 의하여 바뀌워 져야 한다. ext2fs서고는 여러개의 조작클라스의 접근방법을 제공해 준다.

첫번째 조작클라스는 파일체계지향조작들이다. 프로그람이 파일체계를 열거나 닫을수 있고 비트매프자료를 읽거나 쓸수도 있으며 또 디스크상에 새로운 파일체계를 생성할수도 있다. 함수들은 또한 파일체계의 불량블로크목록을 관리할수도 있다.

두번째 조작클라스는 등록부에 영향을 주는 클라스들이다. ext2fs서고 호출자는 등록부를 생성하고 확장할수 있을뿐아니라 등록부입구점을 첨가하거나 삭제할수도 있다. 함수들은 색인마디번호에 대한 경로이름을 찾거나 색인마디번호가 주어 질 때 색인마디의 경로이름을 결정하기 위하여 제공된다.

마지막 조작클라스를 리용하여 색인마디의 주사, 읽기, 쓰기 등이 가능하며 색인마디안의 모든 블로크들을 주사할수 있다. 배정과 배정해제부분프로그람도 리용할수 있으며 사용자방식프로그람들의 블로크와 색인마디들을 배정하거나 해제할수 있다.

ext2fs도구

ext2fs에 리용되는 강력한 관리도구들이 개발되여 있다. 이 편의프로그람들은 생성및 수정에 리용되며 ext2파일체계내에서 임의의 불일치성을 교정하는데 리용된다.

mk2fs프로그람은 빈 ext2파일체계를 확보하기 위하여 구획을 초기화하는데 리용된다.

tune2fs프로그람은 파일체계파라메터들을 변경시키는데 리용된다. 이미 개선된 ext2fs 특성에서 설명한것처럼 이 프로그람은 오유상태, 최대올려태우기계수값, 최대검사간격 그리고 상위사용자를 위하여 예약된 론리적블로크들의 수를 변경시킬수 있다. 그러나 가장 흥미 있는 도구는 아마 파일체계검사기일것이다.

e2fsck는 체계의 부정중지후에 파일체계의 불일치성을 해소시키는데 리용된다. e2fsck의 초기판본은 Minix파일체계를 위한 리누스 토발즈의 fsck프로그람에 기초하고 있다. e2fsck의 현재 판본은 ext2fs서고의 스크래치로부터 작성되였으며 초기판본보다도 훨씬 더 빠르고 파일체계의 많은 불일치성을 교정할수 있다. e2fsck프로그람은 가능한대로 빨리 실행할수 있게 설계되였다. 파일체계검사기들은 디스크구역성을 지향하기때문에 e2fsck에 의하여 리용된 알고리듬의 최량화에 의하여 실현되며 따라서 파일체계구조체에는 디스크로부터 반복적으로 접근하지 못한다. 또한 색인마디와 등록부가 검사된 순위는 디스크찾기시간을 줄이기 위하여 블로크번호에 따라 정렬된다.

통과 1에서 e2fsck는 파일체계안의 전체 색인마디에 걸쳐 반복하며 파일체계의 비련결객체에 관하여 매 색인마디를 검사한다. 이 검사는 다른 파일체계객체에 대하여 그 어떤 교차검사를 요구하지 않는다. 통과 1이 실행되는 기간 어느 블로크들과 색인마디들이 사용중에 있는가를 지적하는 비트매프들이 콤파일된다. 만일 e2fsck가 하나이상의 색인마디가 자료블로크들을 요구한다면 매개 색인마디가 공유블로크의 자체복사를 가지도록 공유블로크를 보조하거나 혹은 하나이상의 색인마디를 배정해제시키는 방법으로 이러한 모순을 해결하기 위하여 1B로부터 1D로 통과하게 된다. 모든 색인마디들에 대하여 기억으로의 읽기와 검사가 진행되여야 하므로 통과 1은 실행시간이 오래 걸린다. 다음 통과들에서 I/O시간을 줄이기 위하여 림계적인 파일체계정보가 캐쉬에 기억된다.

이 기술에 대한 가장 중요한 실례는 파일체계상의 모든 등록부블로크의 디스크우에서의 위치이다. 이 조작으로 하여 해당 정보를 얻는데서 통과 2가 실행되는 동안 등록부색인마디구조체를 다시 읽어야 할 필요가 없어 지게 된다.

통과 2는 비련결객체로서 등록부를 검사한다. 등록부입구점들이 디스크블로크들을 련결하지 않기때문에 매개 등록부블로크는 다른 등록부블로크를 참조하지 않고 개별적으로 검사할수 있다. 이것은 e2fsck가 모든 등록부블로크들을 블로크번호에 따라 정렬할수 있게 하며 높아 지는 순서로 검사하기때문에 디스크찾기시간을 줄인다. 등록부블로크들은 등록부입구점들이 유효하다는것을 확인하기 위하여 검사되며 또한 사용중에 있는 색인마디번호에 대한 참조값을 포함한다(통과 1에서 결정된것처럼).

매 등록부색인마디안의 첫 등록부블로크에서 “.”과 “..”입구점들은 현재 그것들이 존재한다는것과 “.”입구점의 색인마디번호가 현행등록부와 일치한다는것을 확인하기 위하여 검사된다(“..”입구점에 대한 색인마디번호는 통과 3까지 검사되지 않는다.).

통과 2에서는 또한 매 등록부가 련결된 상위등록부와 관련이 있는 정보를 캐쉬에 기억한다. 만약 등록부가 한개이상의 등록부에 의하여 참조되면 등록부에 대한 두번째 참조는 부정확한 hard link로 취급되여 제거된다.

통과 2의 끝에서 e2fsck의 조작수행에 필요되는 모든 디스크 I/O가 거의 완성된다는데 대하여 강조한다.

통과 3, 4, 5가 요구하는 정보는 캐쉬에 기억되기때문에 e2fsck의 나머지 통과조작들

은 CPU에 크게 속박되고 전체 실행시간의 3~5%보다 더 작아 진다.

통과 3에서 등록부의 련결성이 검사된다. e2fsck는 통과 2에서 캐쉬에 기억된 정보를 리용하여 뿌리까지 매 등록부의 경로를 거꾸로 추적한다.

이때 "..”입구점도 역시 유효성을 확인하기 위하여 검사되며 뿌리까지 거꾸로 추적할수 없는 임의의 등록부들은 /lost+found등록부와 련결된다.

통과 4에서 e2fsck는 색인마디전체에 대하여 반복동작으로 모든 색인마디들의 참조값을 검사하며 통과 2와 3의 실행기간에 계산된 내부계수값을 련결계수값(통과 1에서 캐쉬에 기억된)과 비교하는 방법으로 전체 색인마디의 참조값을 검사한다.

이 통과기간 련결계수값이 령인 지워 지지 않은 임의의 파일들은 역시 /lost+found등록부에 련결된다.

끝으로 통과 5에서 e2fsck는 파일체계의 요약정보의 유효성을 검사한다. 검사는 블로크와 파일체계상의 실제적비트매프에 대하여 선행통과과정에 구성된 색인마디비트매프와 비교하는 방법으로 실현된다. 또한 필요한 경우 디스크상의 복사조작으로 수정한다.

파일체계오유수정프로그람(debuger)도 또 하나의 쓸모 있는 도구이다.

Debugfs는 파일체계의 상태를 검열하거나 변화시키는데 리용되는 강력한 프로그람이다.

기본적으로 이 프로그람은 ext2fs서고와의 호상작용대면부를 제공한다. 사용자에 의하여 건입력된 명령은 서고부분프로그람에 대한 호출로 변화된다. debugfs는 파일체계의 내부구조를 검열하는데 리용될수 있는데 손상된 파일전체를 수동적으로 복구하거나 혹은 e2fsck를 위한 검열실례를 생성한다. 이 프로그람은 그것이 무엇을 하는지 모르는 사람들이 리용하면 위험할수 있다. 왜냐하면 이 도구를 잘못 리용하여 파일체계를 쉽게 파괴할수 있기때문이다. 이런데로부터 기정값은 debugfs가 파일체계를 읽기전용방식으로 열수 있게 설정한다. 사용자는 debugfs가 파일체계를 읽기/쓰기접근방식으로 열도록 하기 위하여 _w기발을 명백하게 정의하여 주어야 한다.

2차확장파일체계는 Linux용의 확장성 있고 강력한 파일체계로 연구되였다. 이 파일체계는 현재까지 Linux계에서 가장 성과적인 파일체계이며 현재 발송되는 Linux의 모든 배포판들의 기초로 되는 체계이다. 많은 파일체계와 마찬가지로 ext2파일체계는 파일에 보존된 자료가 자료블로크에 보존된다는 전제하에서 만들어 진다. 이 자료블로크는 모두 길이가 같으며 서로 다른 ext2파일체계사이에서 그 길이가 변할수 있다 해도 특정한 ext2파일체계의 블로크크기는 블로크가 생성될 때 설정된다(mke2fs를 리용하여).

매개 파일의 크기는 블로크옹근수로 둥그리기된다. 만일 블로크의 크기가 1024byte이면 1025byte짜리 파일은 2개의 1024짜리 블로크를 차지하게 된다. 그러나 이것은 파일당 평균적으로 블로크가 절반정도 랑비된다는것을 의미한다. 보통 CPU리용률을 계산하는데서 타협할수 있는 기준은 기억과 디스크의 공간리용성이다. 대다수 조작체계와 함께 Linux의 경우에도 CPU의 작업부하를 줄이기 위하여 무효한 디스크의 리용을 상대적으로 제한하고 있다. 파일체계에서 모든 블로크들이 다 자료를 보존하는것이 아니라 그중 몇개는 파일체계의 구조를 표현하는 정보를 보관하는데 리용된다.

ext2는 파일체계의 매 파일을 색인마디자료구조로 서술하여 파일체계의 기하학적모양을 정의한다. 색인마디는 파일안의 자료가 어느 블로크를 차지하고 있는가를 서술할뿐아니라 파일의 접근권한, 파일의 변경시간 그리고 파일의 형도 서술한다. ext2파일체계에

서 매 파일은 단일한 색인마디에 의하여 서술되며 매 색인마디는 단일하고 일의적인 식별번호를 가진다.

파일체계에 관한 색인마디들은 모두 색인마디표에 함께 보존된다. ext2등록부들은 해당 등록부입구점들의 색인마디에 대한 지적자를 포함하는 특별한 파일이다(그자체가 색인마디에 의하여 서술된).

그림 7-1은 ext2파일체계에서 블로크구조로 된 장치안의 블로크렬들이 차지하는 상태에 대하여 보여 준다.
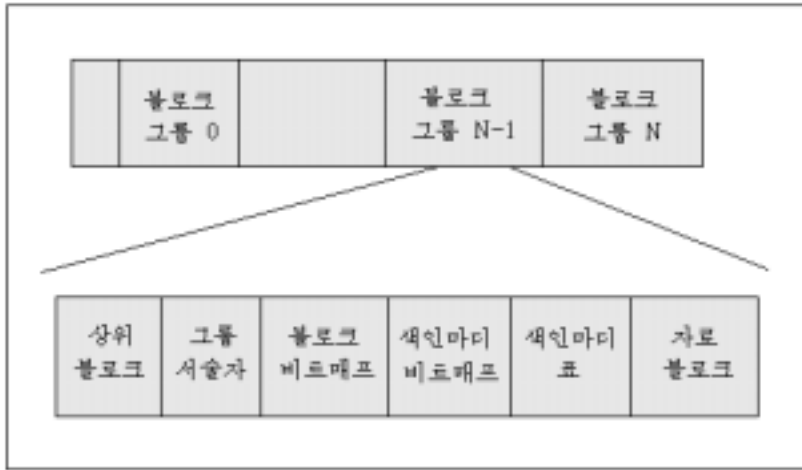
그림 7-1. ext2파일체계의 물리적구조

매 파일체계가 서로 관계되는 한에 있어서는 블로크장치가 곧 읽거나 쓸수 있는 블로크들의 렬로 된다. 파일체계는 물리적매체상에서 블로크가 어디에 놓여야 하며 장치구동기의 일감이 어느것인지에 대하여 알 필요가 없다.

파일체계는 자료를 포함하고 있는 블로크장치로부터 정보나 자료를 읽어야 할 때마다 자기가 지원하는 장치구동기가 옹근수개의 블로크를 읽을것을 요구한다. ext2파일체계는 그것이 차지하는 론리구획을 블로크그룹들로 분할한다. 매 그룹은 파일체계의 완전성에 관한 중대한 정보들뿐아니라 유지하고 있는 실지 파일과 등록부들을 정보와 자료블로크로서 복제한다. 이 복제는 장애가 발생하였거나 파일체계를 복구해야 할 필요가 있는 경우에 필요하다.

다음의 부분내용들은 매개 블로크그룹의 내용을 더 구체적으로 서술한다.

ext2fs의 색인마디

색인마디는 한개의 색인마디에 의하여 서술되는 파일체계안에 매개 파일이나 등록부를 가진 ext2파일체계의 기본구축블로크이다. 매 블로크그룹을 위한 ext2색인마디는 체계가 배정 혹은 배정되지 않은 색인마디를 기억할수 있게 하는 비트매프와 함께 색인마디표에 보존된다. 그림 7-1은 ext2의 색인마디양식을 보여 준다.

색인마디는 다음에 설명하는 마당들을 포함한다.

방 식

방식은 두개의 정보단위 즉 색인마디가 무엇을 서술하는가와 사용자가 거기에 보존해야 할 허가권을 기억한다. ext2에서 색인마디는 파일, 등록부, 기호련결, 블로크장치, 물리장치, 대기렬 등을 서술할수 있다.

소유자정보

이 마당은 파일이나 등록부소유자들에 대한 사용자와 그룹의 식별자들을 포함한다.

크 기

바이트수로 정의되는 파일의 크기

시간표식

색인마디가 생성된 시간과 그것이 변경된 마지막시간

자료블로크

색인마디가 서술하고 있는 자료를 포함하는 블로크에 대한 지적자들중 첫 12개는 색인마디에 의하여 서술된 자료를 포함하는 물리적블로크들에 대한 지적자이며 마지막 3개 지적자는 보다 더 높은 간접준위를 포함한다. 실례로 2중간접블로크지적자는 자료블로크를 지적하는 블로크지적자에서 지적된다. 이것은 자료블로크의 길이가 12보다 작거나 같은 파일들이 그것보다 큰 파일들보다 더 빨리 접근한다는것을 의미한다. ext2색인마디들은 전용장치파일들을 서술할수 있는데 이 장치파일들은 실지 파일은 아니고 프로그람들이 장치에 접근하는데 사용될수 있는 핸들(조종자)이라는데 대하여 강조한다.
/dev안의 모든 장치파일들은 프로그람들이 Linux의 장치에 접근하게 한다. 실례로 올려태우기프로그람은 올려태우기해야 할 장치를 한개의 인수로 취한다.

ext2fs의 상위블로크

상위블로크는 파일체계의 기본크기와 모양을 서술한다. 상위블로크안에 있는 정보는 파일체계관리기가 파일체계를 관리하고 리용할수 있게 한다. 보통 체계가 올려태우기될 때에는 블로크그룹 0에 있는 상위블로크만을 읽지만 매 블로크그룹은 파일체계가 손상되는 경우에 2중복사본을 포함한다.

상위블로크는 다음과 같은 정보들을 포함한다.

매지크번호

매지크(magic)번호는 프로그람올려태우기동작시 해당 프로그람이 실지로 ext2파일체

계의 상위블로크인가를 검사하는데 리용된다. 현재의 ext2판본에서는 이 값이 OxEF53이다.

개정준위

기본개정준위와 부차개정준위는 올려태우기코드가 파일체계지원특성을 파일체계의 특정한 개정준위에서만 리용할수 있는가에 대하여 판정할수 있게 한다. 여기에는 올려태우기코드가 현재 파일체계에서 어떤 새로운 특성을 안전하게 리용할수 있는가를 결정하게 하는 특성호환마당들이 있다.

올려태우기계수값과 최대올려태우기계수값

파일체계가 완전히 검사되였는가를 결정하는데 이 정보들을 리용한다. 올려태우기계수값은 매번 체계가 올려태우기될 때마다 증가되며 그 계수값이 최대올려태우기계수값과 같아 지면 경고통보문“maximum mount count reached, running e2fsck is recommended”를 현시한다.

블로크그룹번호

상위블로크의 복사를 보존하고 있는 블로크그룹의 번호이다.

블로크크기

바이트수로 표시되는 파일체계에서 블로크의 크기. 실례로 1024byte이다.

그룹당 블로크수

그룹안에서의 블로크수는 블로크크기와 마찬가지로 파일체계가 생성될 때 고정된다.

자유블로크

파일체계안의 자유색인마디들의 수이다.

첫번째 색인마디

파일체계안의 첫번째 색인마디의 번호. ext2뿌리파일체계에서 첫번째 색인마디는 “/ ”등록부에 대한 입구점들로 된다.

ext2그룹서술자

매개 블로크서술자들은 그것을 서술하는 자료구조체를 가지고 있다. 상위블로크에서처럼 모든 블로크그룹들에 관한 전체 그룹서술자는 파일체계가 파손될 때 매개 블로크그룹에 복제된다. 매개의 그룹서술자는 다음정보를 포함한다.

블로크비트매프

블로크그룹에 대한 블로크배정 비트매프의 블로크번호이다. 이것은 블로크의 배정 및 배정해제시에 리용된다.

색인마디비트매프

블로크그룹에 대한 색인마디배정비트매프의 블로크번호이다. 역시 색인마디의 배정과 배정해제에 리용된다.

색인마디표

해당 블로크그룹의 색인마디표에 대한 출발블로크의 블로크번호이다. 매 색인마디는 아래에 서술한 ext2색인마디자료구조체에 의하여 서술된다.

자유블로크계수, 자유색인마디계수, 리용된 등록부계수

그룹서술자가 차례로 배치되여 그룹서술자표를 구성한다. 매 블로크그룹은 전체 그룹서술자표를 포함하며 그뒤에 상위블로크의 복제부분이 포함된다.

첫번째 복사부분(블로크그룹 0의)만이 실제적으로 ext2파일체계에서 리용된다. 기본복사부분이 손상된 경우에 상위블로크의 복사부분과 같이 다른 복사부분들은 거기에 남아 있게 된다. ext2파일체계에서 등록부들은 파일체계안의 파일에 대한 접근경로를 생성하고 유지하는데 리용되는 특별한 파일이다. 등록부파일은 매개가 다음정보를 포함하고 있는 등록부입구점들의 목록이다.

색인마디

등록부입구점에 대한 목록이다. 이것은 블로크그룹의 색인마디표에 기억된 색인마디 배렬첨수이다. 실례로 "file"이라는 이름을 가진 파일에 대한 등록부입구점은 색인마디번호 il에 대한 참조값을 가지고 있다.

이름길이

바이트수로 표시되는 등록부입구점의 길이이다.

이 름

등록부입구점의 이름이다. 매 등록부의 첫 두개의 입구점은 항상 "this directory"와 "parent directory"를 의미하는 표준 "."과 ".."입구점이다.

ext2파일체계에서 파일의 크기변경

파일체계에서의 공통적인 문제점은 그것의 토막화경향성이다. 파일의 자료를 보관하는 블로크들이 파일체계전반에 널리 퍼져 자료블로크들사이의 거리가 멀어짐으로써 파일자료블로크에 대한 순차적접근은 점점 더 어려워 지게 된다. ext2파일체계는 이 문제를

현재 자료블로크들과 물리적으로 가까운 파일 혹은 적어도 같은 블로크그룹에 새 블로크를 배정하는 방법으로 극복하고 있다. 이것이 실패할 때에만 다른 잠금그룹에 자료블로크를 배정한다. 어떤 프로쎄스가 파일에 자료를 쓰려고 할 때마다 Linux파일체계는 자료가 파일의 마지막 배정블로크의 끝에서 제거되였는가를 알아 보기 위하여 검사를 진행한다. 만일 제거되였다면 파일에 새로운 자료블로크를 배정해야 한다. 배정이 완료될 때까지 프로쎄스는 실행될수 없다. 프로쎄스는 새로운 자료블로크를 배정하기 위하여 파일체계를 기다려야 하며 그것을 계속하기에 앞서 자료블로크에 자료의 나머지를 써넣어야 한다. ext2블로크배정프로그람이 실행하는 첫번째 과제는 파일체계의 ext2상위블로크를 잠그는것이다. 배정과 배정해제는 상위블로크안의 마당들을 변화시키는데 Linux파일체계는 하나이상의 프로쎄스가 동일한 시간에 이 동작을 수행하게 할수 없다. 만일 다른 프로쎄스가 더 많은 자료블로크를 배정하려고 한다면 해당 프로쎄스가 완료될 때까지 대기해야 할것이다. 상위블로크를 기다리는 프로쎄스는 중단되며 상위블로크의 조종이 현행사용자에 의하여 제거될 때까지 실행할수 없다. 상위블로크에 대한 접근은 먼저 온 객체에게 먼저 봉사하는 방식에 기초하여 부여되며 일단 프로쎄스가 상위블로크의 조종에 착수하면 완료될 때까지 조종을 계속한다. 상위블로크의 잠금이 완료된후 프로쎄스는 파일체계에 자유블로크가 충분히 남아 있는가를 검사한다. 자유블로크가 충분하지 못하면 배정을 더하려던 과정은 실패하며 프로쎄스는 파일체계상위블로크의 조종을 철회한다. 만약 자유블로크가 충분하게 있으면 프로쎄스는 그것을 배정하게 된다. ext2파일체계가 자료블로크들을 선행배정할수 있게 구성되여 있으면 그 블로크들중의 어느 하나를 사용할수 있다. 선행배정된 블로크들이 실제적으로 존재하지 않으면 배정된 블로크비트매프안에 예약된다. 자료블로크를 배정하려고 하는 파일을 표현하는 VFS의 색인마디는 두개의 ext2정의용마당 prcalloc_block와 prealloc_count를 가지고 있는데 이것들은 각각 첫번째 선행배정 자료블로크번호와 그것들의 개수이다. 선행배정된 블로크가 전혀 없거나 블로크선행배정이 불가능하면 ext2파일체계는 새 블로크를 배정해야 한다. ext2파일체계는 우선 파일안에 새 자료블로크가 비여 있는가를 알아 본다. 론리적으로 이것은 순차접근을 더 빨리 실현할수 있기때문에 배정에서 제일 효과적인 블로크로 된다. 이 블로크가 비여 있지 않으면 탐색공간을 더 넓혀 리상적으로 64개블로크내에서 자료블로크를 찾는다. 리상적이 아니라고 해도 이 블로크는 아주 가까이에 있으며 적어도 파일에 속하는 다른 자료블로크들의 같은 블로크그룹내에 있다. 만약 다음 블로크가 비여 있지 않으면 프로쎄스는 여러개의 자유블로크들을 찾을 때까지 차례로 다른 모든 블로크그룹들을 찾기 시작한다. 블로크배정코드는 한개의 블로크그룹안의 어디에서든지 8개의 빈 자료블로크의 무리 짓기(cluster)를 찾아 낸다. 만약 8개를 동시에 찾지 못하면 적은 량이라도 받아 들인다. 또한 블로크선행배정이 요구되고 가능하면 prealloc_block와 prealloc_count를 적당히 갱신한다. 자유블로크들을 찾아 낼 때마다 블로크배정코드가 블로크그룹의 블로크비트매프를 갱신하며 캐쉬에 자료완충기를 배정한다. 이 자료완충기는 파일체계가 지원하는 장치식별자와 배정된 블로크의 블로크번호에 의하여 일의적으로 식별된다.

완충기안의 자료는 값이 령이며 이때 물리적디스크에 씌여 지지 않은 내용이라는 것을 보여 주기 위하여 "dirty"라고 표식을 단다. 끝으로 상위블로크 그자체도 변경되지 않고 잠금이 되지 않았다는것을 보여 주기 위하여 "dirty"로 표시한다. 만일 상위블로크를 기다리는 어떤 프로쎄스들이 있을 때는 대기렬의 첫번째 프로쎄스에 대하여 실행이 다시 허용되며 파일의 조작과 관련한 배타조종이 이루어 지게 된다. 프로쎄스의 자료는 새 자료블로크에 기록되며 만일 자료블로크가 채워 져 있으면 전체 프로쎄스가 반복되여 다른 자료블로크가 배정된다 .이 부분에서는 상위블로크의 배치구성을 서술한다.

아래의 내용은 ext2fs상위블로크 [include/linux/ext2_fs.h]의 공인된 구조체이다.

```
struct ext2_super_block{
   unsigned long   s_inodes_count;
   unsigned long   s_blocks_count;
   unsigned long   s_r_blocks_count;
   unsigned long   s_free_blocks_count;
   unsigned long   s_free_inodes_count;
   unsigned long   s_first_data_block;
   unsigned long   s_log_block_size;
            long   s_log_frag_size;
   unsigned long   s_block_per_group;
   unsigned long   s_frags_per_group;
   unsigned long   s_inodes_per_group;
   unsigned long   s_mtime;
   unsigned long   s_wtime;
   unsigned short  s_mnt_count;
            short  s_max_mnt_count;
   unsigned short  s_magic;
   unsigned short  s_state;
   unsigned short  s_errors;
   unsigned short  s_pad;
   unsigned long   s_lastcheck;
   unsigned long   s_checkinterval;
   unsigned long   s_reserved[238];
   };
```

아래에 몇가지 설명을 준다.

s_inode_count	fs(파일체계)상의 색인마디의 총수
s_block_count	fs상의 블로크의 총수
s_r_block_count	상위블로크의 배타리용에 예약된 블로크의 총수
s_free_blocks_count	fs상의 자유블로크의 총수
s_free_inodes_count	fs상의 자유색인마디의 총수
s_first_data_block	첫 블로크의 fs상에서의 위치. 보통 이 항목은 1024byte의 블로크를 포함하는 fs에서는 블로크번호 1이며 다른 fs에서는 번호 0이다.
s_log_block_size	론리적블로크 크기를 바이트로 계산하는데 리용.론리적블로크 크기는 1024≦s_log_block_size이다.
s_log_frag_size	론리토막의 크기를 계산하는데 리용. 론리토막의 크기는 s_log_frag_ size가 정수이면 1024≦s_log_ frag_size이고 부수이면 1024≧s_log_ frag_size이다.
s_blocks_per_group	한개 그룹에 포함된 블로크의 총수
s_frags_per_group	한개 그룹에 포함된 토막들의 총수
s_inodes_per_group	한개 그룹에 포함된 색인마디들의 총수
s_mtime	fs가 마지막으로 올려태우기된 시간
s_wtime	fs상의 상위블로크의 마지막쓰기가 수행된 시간
s_mnt_count	검사를 진행하지 않고 fs가 읽기/쓰기방식으로 올려태우기된 시간
s_max_mnt_count	검사하기전에 fs가 읽기/쓰기방식으로 올려태우기되는 총 시간
s_magic	파일체계에 대한 식별을 허용하는 매지크수 표준 ext2fs에 대해서는 OxEF53이며 0.2b전의 ext2fs판본에서는 OxEF51이다.
s_state	파일체계의 상태. 명백하게 올려태우기가 해제되였음을 의미하는 EXT2_VALID_FS(Ox0001)과 핵심부코드에 의하여 오유가 검출되였다는것을 의미하는 EXT2_ERROR_FS(Ox0002)의 론리합
s_error	오유가 발생할 때 어떤 동작을 수행하겠는가를 지적한다.
s_pad	리용되지 않는다.
s_lastcheck	파일체계상에서 수행된 마지막검사시간
s_checkinterval	파일체계상에서 검사들사이의 최대가능한 시간
s_reserved	리용되지 않는다.

시간들은 1970.1.1 GMT 00 : 00 : 00으로부터 초단위로 측정된다.

일단 상위블로크가 기억에 적재되면 ext2fs핵심부코드는 일련의 다른 정보들을 계산하며 그것을 다른 구조체에 보존한다.

이 구조체는 다음의 형식을 가진다.

```
struct ext_sb_info{
          unsigned long s_frag_size;
          unsigned long s_frags_per_block;
          unsigned long s_inodes_per_block;
          unsigned long s_frags_per_group;
          unsigned long s_block_per_group;
          unsigned long s_inodes_per_group;
          unsigned long s_itb_per_block;
          unsigned long s_groups_count;
          struct buffer_head * s_sbh;
          struct ext2_super_block*s_es;
          structbuffer_head *s_group_desc[EXT2_MAX_GROUP_
                          DESC];
          unsigned short    s_loaded_inode_bitmap;
          unsigned short    s_loaded_block_bitmaps;
          unsigned long      s_inode_bitmap_number[EXT2_
                          MAX_ GROUP_LOADED];
         struct buffer_head   *s_block_bitmap[EXT2_
                          MAX_GROUP_LOADED];
         unsigned long       s_block_bitmap_number
                          [EXT2_ MAX_GROUP_LOADED];
         int s_rename_lock;
         struct wait_queue   *s_rename_wait;
         unsigned long       s_mount_opt;
         unsigned short      s_mount_state;
          };
```

구조체의 성분들에 대한 설명을 다음에 준다.

s_frag_size	바이트수로 된 토막의 크기
s_frags_per_block	한개 블로크안의 토막의 수
s_inodes_per_block	색인마디표블로크안에서의 색인마디의 수
s_frags_per_group	그룹내에서 토막의 수
s_inodes_per_group	그룹안에서의 색인마디의 수
fs_blocks_per_group	그룹내에서의 블로크의 수
s_itb_per_group	매 블로크당 그룹서술자의 수
s_groups_count	그룹의 수
s_sbh	기억안의 디스크상위블로크를 포함하는 완충기
s_es	완충기의 상위블로크에 대한 지적자
s_group_desc	그룹서술자를 포함하는 완충기에 대한 지적자
s_loaded_inode_bitmaps	리용된 색인마디비트매프의 캐쉬입구점들의 수
s_loaded_inode_bitmap	리용된 블로크비트매프의 캐쉬입구점 들의 수
s_inode_bitmap_number	완충기안의 색인마디비트매프가 어느 그룹에 속하는가를 지적
s_block_bitmap	블로크비트매프의 캐쉬
s_rename_lock	파일체계에서 두개의 동시적인 이름재정의 조작을 피하기 위하여 리용되는 잠금
s_rename_wait	프로쎄스에 이름재정의조작이 끝나기를 기다리는데 리용하는 대기렬
s_mount_opt	관리기에 의하여 정의되는 올려태우기선택
s_mount_state	디스크의 상위블로크로부터 계산되는 대다수 상태들의 값

Linux ext2fs관리기는 색인마디와 블로크비트매프들에 대한 접근을 고속완충기억한다. 이 캐쉬는 맨 마지막에 리용된 완충기와 맨 처음에 리용된 완충기들사이에서 순서화된 완충기들의 목록이다. 관리기는 같은 종류의 비트매프의 고속완충기억방법이나 혹은 디스크에 대한 접근시간을 개선하기 위한 어떤 다른 류사한 방법을 리용해야 한다.

그룹서술자

디스크상에서 그룹서술자는 상위블로크의 바로 뒤에 놓이며 매 서술자는 다음과 같은 형식을 가진다.

```
start ext2_group_desc
{
  unsigned long bg_block_bitmap;
  unsigned long bg_inode_bitmap;
  unsigned long bg_inode_table;
  unsigned long bg_free_block_count;
  unsigned long bg_free_inodes_count;
  unsigned long bg_used_dirs_count;
  unsigned long bg_pad;
  unsigned long bg_reserved[3];
  };
```

여기서 이 구조체의 몇가지 성분들에 대하여 설명한다.

bg_block_bitmap	그룹에 관한 블로크비트매프블로크를 지적
bg_inode_bitmap	그룹에 관한 색인마디비트매프블로크를 지적
bg_inode_table	색인마디표의 첫 블로크 지적
bg_free_inode_count	그룹안의 자유블로크수
bg_free_inode_count	그룹안의 자유색인마디수
bg_used_dirs_count	그룹안의 등록부에 배정된 색인마디수
bg_pad	패딩

그룹서술자내의 정보는 오직 그것이 서술하고 있는 그룹에만 실제적으로 관련된다.

비트매프

ext2파일체계는 배정된 블로크들과 색인마디의 위치를 항시적으로 탐색하기 위하여 비트매프를 리용한다. 매 그룹의 블로크비트매프는 그룹안의 첫 블로크로부터 마지막블로크사이의 범위에 있는 블로크들을 참조한다. 정확한 블로크의 비트로 접근하기 위하여 먼저 블로크가 속하는 그룹을 찾고 다음 그 그룹에 포함되는 블로크비트매프의 블로크비트를 찾는다. 사실 블로크비트매프가 파일체계에 의하여 지원되는 가장 작은 배정단위를 토막(fragment)이라고 간주하는것은 아주 중요한 리해로 된다. 블로크크기가 항상 토막크기들의 중복으로 이루어 지기때문에 파일체계관리기는 블로크를 배정할 때 실제적으로 다중화된 개수의 토막을 배정한다. 이 블로크비트매프의 리용은 파일체계관리기로 하여금 토막에 기초하여 공간을 배정하거나 배정해제할수 있게 한다. 매 그룹색인마디비트매프는 그룹의 처음부터 마지막색인마디까지의 범위에 있는 색인마디들을 참조한다. 정확

한 색인마디의 비트에 접근하기 위하여 먼저 색인마디가 속한 그룹을 찾고 다음에 그 그룹안에 포함된 색인마디비트매프의 색인마디비트를 찾는다. 색인마디표로부터 색인마디정보를 얻는 방법은 마지막탐색이 색인마디비트매프대신 그룹색인마디표에서 진행된다는 것을 제외하고는 토막에 대하여 진행하는 방법과 꼭 같다.

색인마디

색인마디는 파일을 유일적으로 서술한다.
색인마디구조체의 형식을 아래에 보여 주었다.

```
struct ext2_inode    {
  unsigned short  i_mode;
  unsigned short  i_uid;
  unsigned long   i_size;
  unsigned long   i_atime;

  unsigned long   i_ctime;
  unsigned long   i_mtime;
  unsigned long   i_dtime;
  unsigned short  i_gid;
  unsigned short  i_links_count;
  unsigned long   i_blocks;
  unsigned long   i_flags;
  unsigned long   i_reserved1;
  unsigned long   i_block[ext2_n_blocks];
  unsigned long   i_version;
  unsigned long   i_file_acl;
  unsigned long   i_dir_acl;
  unsigned long   i_faddr;
  unsigned char   i_frag;
  unsigned char   i_fsize;
  unsigned short  i_pad1;
  unsigned long   i_reserved2[2];
 };
```

몇가지 구조체성분에 대하여 서술한다.

i_mode	파일형(문자, 블로크, 련결 등)과 그 파일에 대한 접근권한
i_uid	파일소유자의 Uid
i_size	바이트단위의 론리크기
i_atime	파일에 접근된 마지막 시간
i_ctime	파일의 색인마디정보가 변경된 마지막시간
i_mtime	파일내용을 변경시킨 마지막시간
i_dtime	파일이 지워 질 때
i_gid	파일의 gid
i_links_count	이 파일을 지적하는 련결의 수
i_block	512byte단위로 계수되는 파일에 배정된 블로크의 수
i_flags	기발(아래를 볼것)
i_reserved1	예약
i_block	블로크에 대한 지적자(아래를 볼것)
i_version	파일의 판본(NFS에서 리용)
i_file_acl	파일의 접근조종목록(아직 리용되지 않음)
i_dir_acl	등록부의 접근조종목록(아직 리용되지 않음)
i_faddr	파일의 토막들이 존재하는 블로크
i_size	토막의 크기
i_pad1	패딩
i_reserved2	예약

보는바와 같이 색인마디는 블로크에 대한 EXT2_N_BLOCKS(ext2fs 0.5에서는 15)지적자를 포함하고 있다. 이 지적자들중에서 첫 EXT2_N_BLOCKS(12)는 자료에 대한 직접지적자이다.

다음의 입구점은 자료에 대한 지적자들의 블로크를 지적한다(간접). 그 다음의 입구점은 자료에 대한 지적자들의 블로크에 대한 지적자블로크를 지적한다(2중간접).

다음입구점은 자료지적자블로크에 대한 지적자들의 블로크를 지적하는 지적자블로크를 지적한다(3중간접).

색인마디기발들은 다음과 같은 하나이상의 론리합을 취할수 있다.

EXT2_SECRM_FL 0x0001

안전한 지우기 이 조작은 보통 이 기발이 설정될 때 파일을 지운다는것을 의미한다. 우연자료가 파일에 이미 배정된 블로크들에 씌여 질수 있다.

EXT2_UNRM_FL_0x0002

지우기해제 이 기발이 설정되고 파일이 지워 지고 있을 때를 의미하며 이때 파일체계코드는 파일의 지우기해제를 확인할수 있는 충분한 정보를

기억해야 한다(일정한 내용으로).

EXT2_COMPR_FL_0x0004

압축파일 파일의 내용이 압축된다. 파일체계코드는 이 파일에 접근할 때 반드시 압축/압축해제알고리듬을 리용해야 한다.

EXT2_SYNC_FL_0x0008

동기갱신 이 파일디스크표시는 내부중심파일디스크표시와 동기를 유지해야 한다. 이런 종류의 파일상에서 비동기적I/O는 불가능하다. 동기갱신은 색인마디자체나 간접블로크에만 적용한다. 자료블로크들은 항상 디스크에 비동기적으로 기록된다.

일부 색인마디들은 특별한 의미를 가지고 있다.

EXT2_BAD_INO 1	파일체계의 불량블로크를 포함하는 파일
EXT2_ROOT_INO 2	파일체계의 뿌리등록부
EXT2_ACL_IOX_INO 3	ACL색인마디
EXT2_ACL_DATA_INO 4	ACL색인마디
EXT2_BOT_LOADER_INO 5	기동적재기를 포함하는 파일(아직 사용되지 않음)
EXT2_UNDEL_DIR_INO 6	체계가 지워 지지 않는 파일
EXT2_FIRST_INO 11	특별한 의미를 가지지 않는 첫번째 색인마디

등 록 부

등록부는 디스크상의 파일에 대한 접근경로를 생성하는데 리용되는 특별한 파일이다. 색인마디가 많은 접근경로를 가지고 있다는것을 리해하는것은 아주 중요하다. 등록부는 파일체계의 본질적인 부분이기때문에 특별한 구조를 가지고 있다. 등록부파일은 다음의 형식을 가지는 입구점들의 목록이다.

```
start ext2_dir_entry{
unsigned long inode;
unsigned short rec_len;
unsigned short name_len;
Char          name[EXT2_NAME_LEN];
};
```

우의 구조체의 성분들에 대하여 설명하겠다.

inode	파일의 색인마디에 대하여 지적한다.
rec_len	입구점레코드의 길이
name_len	파일이름의 길이
name	파일의 이름. 이 이름은 EXT2_NAME_LEN바이트의 최대길이를 가질수도 있다(판본 0.5에서처럼 255).

등록부안의 매개 파일에 대하여 등록부파일입구점이 존재한다. ext2fs는 UNIX파일체계이므로 등록부안의 첫 두개 입구점은 현행등록부와 그것의 상위등록부를 지적하는 "." 과 ".." 파일이다.

배정알고리듬

다음의 내용은 ext2파일체계관리기가 리용할 배정알고리듬이다. 현재 많은 사용자들이 동일한 콤퓨터상에서 한개이상의 조작체계를 리용하고 있다.

만일 동일한 ext2구획을 한개이상의 조작체계가 리용한다면 동일한 배정알고리듬을 리용해야 한다. 또한 다른 알고리듬을 리용한다면 한 파일체계관리기가 다른 파일체계관리기의 동작을 망치게 할수 있다.

또한 다른 파일체계관리기가 배정에 피해를 주지 않고 또 간략한 알고리듬을 리용한다면 고도로 효과 있는 배정알고리듬을 리용하는 관리기를 가지고 있어도 얼마 쓸모가 없다.

아래에 새로운 색인마디를 배정하는데 리용되는 규칙들을 제시한다.

▼ 파일의 색인마디는 그것의 상위등록부의 같은 색인마디그룹에 배정된다.
▲ 색인마디들은 그룹들속에 균등하게 배정된다.

새 블로크를 배정하는데 리용되는 규칙들은 다음과 같다.

▼ 새 블로크는 색인마디처럼 같은 그룹에 배정된다.
▲ 련속적인 블로크렬을 배정한다.

물론 이러한 규칙대로 하는것이 불가능한 경우도 있을수 있다. 그러한 경우에 관리기는 블로크나 색인마디중 어디에나 배정할수 있다.

오 유 처 리

이 부분에서는 표준 ext2파일체계가 오유를 어떻게 처리하는가 하느것을 서술한다. 상위블로크는 오유처리방법을 관리하는 두가지 파라메터를 가지고 있다.

첫번째는 기억의 상위블로크구조체안의 s_mount_opt성원이다. 이 값은 파일체계가 올려태우기될 때 정의되는 선택항목으로부터 계산된다.

오유처리와 관련되는 값들은 다음과 같다.

EXT2_MOUNT_ERRORS_CONT	오유가 생겨도 계속한다.
EXT2_MOUNT_ERRORS_RO	파일체계를 읽기방식으로 재올려태우기한다.
EXT2_MOUNT_ERRORS_PANIC	오유에 의한 핵심부의 패니크(위기)

두번째 파라메터는 디스크의 상위블로크구조체의 s_errors성원이다. 이 파라메터는 다음의 값들중에서 어느 하나를 취한다.

EXT2_ERRORS_CONTINUE	오유가 발생한다고 해도 계속한다.
EXT2_ERRORS_RO	파일체계를 읽기전용방식으로 재올려태우기한다.
EXT2_ERRORS_PANIC	핵심부의 단순한 패니크(위기)
EXT2_ERRORS_DEFAULT	기정동작을 사용한다(0.5a의 EXT2_ERRORS_CONTINUE와 같이).

s_mount_opt는 s_errors보다 우선권을 가진다.
다음의것들은 선택항목들의 목록이다.

bsddf	(*)'df '가 bsd처럼 동작하게 한다.
minixdfmakes	'df '는 minix처럼 동작한다.
check=normal	(*)파일체계에 대한 표준검사 수행
check=strict	파일체계에 대한 여유검사 수행
debug	개발자들에게만 허용
errors=contime	(*)파일체계오유를 계수검사
errors=panic	오유가 발생하면 콤퓨터를 정지, 패니크
grpid,bsdgroups	객체들에 상위에 관한 동일한 그룹과 id를 부여
nogrpid,sysvgroup	(*)새 객체들이 생성자의 그룹 id를 가진다.
resuid=n	예약블로크를 리용할수 있는 사용자
resgid=n	예약블로크를 리용할수 있는 그룹
sb=n	이 위치에서 다른 상위블로크를 리용
grpquota,noquota, quota,usrquota	배정량 선택항목들이 ext2에 의하여 무시된다.

원천코드 include/linux/ext2_fs.h

```
/*
 *  linux/include/linux/ext2_fs.h
```

```
 *
 * Copyright (C) 1992, 1993, 1994, 1995
 * Remy Card (card@masi.ibp.fr)
 * Laboratoire MASI - Institut Blaise Pascal
 * Universite Pierre et Marie Curie (Paris VI)
 *
 *  from
 *
 *  linux/include/linux/minix_fs.h
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

#ifndef _Linux_EXT2_FS_H
#define _Linux_EXT2_FS_H

#include <linux/types.h>

/*
 * The second extended filesystem constants/structures
 */

/*
 * Define EXT2FS_DEBUG to produce debug messages
 */
#undef EXT2FS_DEBUG
/*
 * Define EXT2_PREALLOCATE to preallocate data blocks for expanding files
 */
#define EXT2_PREALLOCATE
#define EXT2_DEFAULT_PREALLOC_BLOCKS    8

/*
 * The second extended file system version
 */
#define EXT2FS_DATE           "95/08/09"
#define EXT2FS_VERSION        "0.5b"

/*
```

```
 * Debug code
 */
#ifdef EXT2FS_DEBUG
#     define ext2_debug(f, a...)   { \
                             printk ("EXT2-fs DEBUG (%s, d): %s:",
                             \__FILE__, __LINE__, __FUNCTION__); \printk
                             (f, ## a); \
                             }
#else
#     define ext2_debug(f, a...)   /**/
#endif

/*
 * Special inodes numbers
 */
#define     EXT2_BAD_INO       1    /* Bad blocks inode */
#define EXT2_ROOT_INO          2    /* Root inode */
#define EXT2_ACL_IDX_INO       3    /* ACL inode */
#define EXT2_ACL_DATA_INO      4    /* ACL inode */
#define EXT2_BOOT_LOADER_INO  5    /* Boot loader inode */
#define EXT2_UNDEL_DIR_INO     6    /* Undelete directory inode */

/* First non-reserved inode for old ext2 filesystems */
#define EXT2_GOOD_OLD_FIRST_INO   11

/*
 * The second extended file system magic number
 */
#define EXT2_SUPER_MAGIC     0xEF53

/*
 * Maximal count of links to a file
 */
#define EXT2_LINK_MAX        32000

/*
 * Macro-instructions used to manage several block sizes
 */
#define EXT2_MIN_BLOCK_SIZE             1024
```

```
#define     EXT2_MAX_BLOCK_SIZE           4096
#define EXT2_MIN_BLOCK_LOG_SIZE           10
#ifdef __KERNEL__
# define EXT2_BLOCK_SIZE(s)       ((s)->s_blocksize)
#else
# define EXT2_BLOCK_SIZE(s)       (EXT2_MIN_BLOCK_SIZE << (s)-
                                  >s_log_block_size)
#endif
#define EXT2_ACLE_PER_BLOCK(s)          (EXT2_BLOCK_SIZE(s) / sizeof
                                        (struct ext2_acl_entry))
#define EXT2_ADDR_PER_BLOCK(s)          (EXT2_BLOCK_SIZE(s) / sizeof
                                        (__u32))
#ifdef __KERNEL__
# define EXT2_BLOCK_SIZE_BITS(s)  ((s)->s_blocksize_bits)
#else
# define EXT2_BLOCK_SIZE_BITS(s)  ((s)->s_log_block_size + 10)
#endif
#ifdef __KERNEL__
#define EXT2_ADDR_PER_BLOCK_BITS(s) ((s)->u.ext2_sb.s_addr_per_block_bits)
#define EXT2_INODE_SIZE(s)        ((s)->u.ext2_sb.s_inode_size)
#define EXT2_FIRST_INO(s)         ((s)->u.ext2_sb.s_first_ino)
#else
#define EXT2_INODE_SIZE(s)   (((s)->s_rev_level == EXT2_GOOD_OLD_REV) ? \
                                  EXT2_GOOD_OLD_INODE_SIZE : \
                                  (s)->s_inode_size)
#define EXT2_FIRST_INO(s)    (((s)->s_rev_level == EXT2_GOOD_OLD_REV) ? \
                                  EXT2_GOOD_OLD_FIRST_INO : \
                                  (s)->s_first_ino)
#endif

/*
 * Macro-instructions used to manage fragments
 */
#define EXT2_MIN_FRAG_SIZE                1024
#define     EXT2_MAX_FRAG_SIZE            4096
#define EXT2_MIN_FRAG_LOG_SIZE            10
#ifdef __KERNEL__
# define EXT2_FRAG_SIZE(s)        ((s)->u.ext2_sb.s_frag_size)
# define EXT2_FRAGS_PER_BLOCK(s)  ((s)->u.ext2_sb.s_frags_per_block)
```

```
#else
# define EXT2_FRAG_SIZE(s)          (EXT2_MIN_FRAG_SIZE << (s)-
                                    >s_log_frag_size)
# define EXT2_FRAGS_PER_BLOCK(s)    (EXT2_BLOCK_SIZE(s) / EXT2_FRAG_SIZE(s))
#endif

/*
 * ACL structures
 */
struct ext2_acl_header /* Header of Access Control Lists */
{
      __u32 aclh_size;
      __u32 aclh_file_count;
      __u32 aclh_acle_count;
      __u32 aclh_first_acle;
};

struct ext2_acl_entry  /* Access Control List Entry */
{
      __u32 acle_size;
      __u16 acle_perms;/* Access permissions */
      __u16 acle_type; /* Type of entry */
      __u16 acle_tag;  /* User or group identity */
      __u16 acle_pad1;
      __u32 acle_next; /* Pointer on next entry for the */
                       /* same inode or on next free entry */
};

/*
 * Structure of a blocks group descriptor
 */
struct ext2_group_desc
{
      __u32 bg_block_bitmap;      /* Blocks bitmap block */
      __u32 bg_inode_bitmap;      /* Inodes bitmap block */
      __u32 bg_inode_table;       /* Inodes table block */
      __u16 bg_free_blocks_count; /* Free blocks count */
      __u16 bg_free_inodes_count; /* Free inodes count */
      __u16 bg_used_dirs_count;   /* Directories count */
```

```
      __u16 bg_pad;
      __u32 bg_reserved[3];
};

/*
 * Macro-instructions used to manage group descriptors
 */
#ifdef __KERNEL__
# define EXT2_BLOCKS_PER_GROUP(s)  ((s)->u.ext2_sb.s_blocks_per_group)
# define EXT2_DESC_PER_BLOCK(s)      ((s)->u.ext2_sb.s_desc_per_block)
# define EXT2_INODES_PER_GROUP(s)  ((s)->u.ext2_sb.s_inodes_per_group)
# define EXT2_DESC_PER_BLOCK_BITS(s)  ((s)->u.ext2_sb.s_desc_per_block_bits)
#else
# define EXT2_BLOCKS_PER_GROUP(s)  ((s)->s_blocks_per_group)
# define EXT2_DESC_PER_BLOCK(s)          (EXT2_BLOCK_SIZE(s) / sizeof
                                         (struct ext2_group_desc))
# define EXT2_INODES_PER_GROUP(s)  ((s)->s_inodes_per_group)
#endif

/*
 * Constants relative to the data blocks
 */
#define     EXT2_NDIR_BLOCKS       12
#define     EXT2_IND_BLOCK               EXT2_NDIR_BLOCKS
#define     EXT2_DIND_BLOCK              (EXT2_IND_BLOCK + 1)
#define     EXT2_TIND_BLOCK              (EXT2_DIND_BLOCK + 1)
#define     EXT2_N_BLOCKS                (EXT2_TIND_BLOCK + 1)

/*
 * Inode flags
 */
#define     EXT2_SECRM_FL          0x00000001 /* Secure deletion */
#define     EXT2_UNRM_FL           0x00000002 /* Undelete */
#define     EXT2_COMPR_FL          0x00000004 /* Compress file */
#define EXT2_SYNC_FL               0x00000008 /* Synchronous updates */
#define EXT2_IMMUTABLE_FL          0x00000010 /* Immutable file */
#define EXT2_APPEND_FL             0x00000020 /* writes to file may only
append */
#define EXT2_NODUMP_FL             0x00000040 /* do not dump file */
#define EXT2_NOATIME_FL            0x00000080 /* do not update atime */
```

```
/* Reserved for compression usage... */
#define EXT2_DIRTY_FL                   0x00000100
#define EXT2_COMPRBLK_FL                0x00000200 /* One or more compressed
clusters */
#define EXT2_NOCOMP_FL                  0x00000400 /* Don't compress */
#define EXT2_ECOMPR_FL                  0x00000800 /* Compression error */
/* End compression flags --- maybe not all used */
#define EXT2_BTREE_FL                   0x00001000 /* btree format dir */
#define EXT2_RESERVED_FL                0x80000000 /* reserved for ext2 lib */

#define EXT2_FL_USER_VISIBLE            0x00001FFF /* User visible flags */
#define EXT2_FL_USER_MODIFIABLE         0x000000FF /* User modifiable flags */

/*
 * ioctl commands
 */
#define     EXT2_IOC_GETFLAGS             _IOR('f', 1, long)
#define     EXT2_IOC_SETFLAGS             _IOW('f', 2, long)
#define     EXT2_IOC_GETVERSION           _IOR('v', 1, long)
#define     EXT2_IOC_SETVERSION           _IOW('v', 2, long)

/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
      __u16 i_mode;           /* File mode */
      __u16 i_uid;            /* Low 16 bits of Owner Uid */
      __u32 i_size;           /* Size in bytes */
      __u32 i_atime;          /* Access time */
      __u32 i_ctime;          /* Creation time */
      __u32 i_mtime;          /* Modification time */
      __u32 i_dtime;          /* Deletion Time */
      __u16 i_gid;            /* Low 16 bits of Group Id */
      __u16 i_links_count;    /* Links count */
      __u32 i_blocks;         /* Blocks count */
      __u32 i_flags;          /* File flags */
      union {
            struct {
                  __u32  l_i_reserved1;
            } linux1;
```

```
		struct {
			__u32  h_i_translator;
		} hurd1;
		struct {
			__u32  m_i_reserved1;
		} masix1;
	} osd1;				/* OS dependent 1 */
	__u32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
	__u32 i_generation;		/* File version (for NFS) */
	__u32 i_file_acl;		/* File ACL */
	__u32 i_dir_acl;		/* Directory ACL */
	__u32 i_faddr;			/* Fragment address */
	union {
		struct {
			__u8  l_i_frag;	/* Fragment number */
			__u8  l_i_fsize;	/* Fragment size */
			__u16 i_pad1;
			__u16 l_i_uid_high;	/* these 2 fields    */
			__u16 l_i_gid_high;	/* were reserved2[0] */
			__u32 l_i_reserved2;
		} linux2;
		struct {
			__u8  h_i_frag;	/* Fragment number */
			__u8  h_i_fsize;	/* Fragment size */
			__u16 h_i_mode_high;
			__u16 h_i_uid_high;
			__u16 h_i_gid_high;
			__u32 h_i_author;
		} hurd2;
		struct {
			__u8  m_i_frag;	/* Fragment number */
			__u8  m_i_fsize;	/* Fragment size */
			__u16 m_pad1;
			__u32 m_i_reserved2[2];
		} masix2;
	} osd2;				/* OS dependent 2 */
};

#define i_size_high	i_dir_acl
```

```
#if defined(__KERNEL__) || defined(__linux__)
#define i_reserved1    osd1.linux1.l_i_reserved1
#define i_frag         osd2.linux2.l_i_frag
#define i_fsize        osd2.linux2.l_i_fsize
#define i_uid_lowi_uid
#define i_gid_lowi_gid
#define i_uid_high     osd2.linux2.l_i_uid_high
#define i_gid_high     osd2.linux2.l_i_gid_high
#define i_reserved2    osd2.linux2.l_i_reserved2
#endif

#ifdef     __hurd__
#define i_translator   osd1.hurd1.h_i_translator
#define i_frag         osd2.hurd2.h_i_frag;
#define i_fsize        osd2.hurd2.h_i_fsize;
#define i_uid_high     osd2.hurd2.h_i_uid_high
#define i_gid_high     osd2.hurd2.h_i_gid_high
#define i_author osd2.hurd2.h_i_author
#endif

#ifdef     __masix__
#define i_reserved1    osd1.masix1.m_i_reserved1
#define i_frag         osd2.masix2.m_i_frag
#define i_fsize        osd2.masix2.m_i_fsize
#define i_reserved2    osd2.masix2.m_i_reserved2
#endif
/*
 * File system states
 */
#define     EXT2_VALID_FS           0x0001/* Unmounted cleanly */
#define     EXT2_ERROR_FS           0x0002/* Errors detected */

/*
 * Mount flags
 */
#define EXT2_MOUNT_CHECK            0x0001      /* Do mount-time checks */
#define EXT2_MOUNT_GRPID            0x0004      /* Create files with
                                                directory's group */
#define EXT2_MOUNT_DEBUG            0x0008/* Some debugging messages */
#define EXT2_MOUNT_ERRORS_CONT      0x0010      /* Continue on errors */
```

```
#define EXT2_MOUNT_ERRORS_RO       0x0020     /* Remount fs ro on errors
*/
#define EXT2_MOUNT_ERRORS_PANIC    0x0040     /* Panic on errors */
#define EXT2_MOUNT_MINIX_DF        0x0080/* Mimics the Minix statfs */
#define EXT2_MOUNT_NO_UID32        0x0200  /* Disable 32-bit UIDs */

#define clear_opt(o, opt)          o &= ~EXT2_MOUNT_##opt
#define set_opt(o, opt)            o |= EXT2_MOUNT_##opt
#define test_opt(sb, opt)          ((sb)->u.ext2_sb.s_mount_opt & \
                                   EXT2_MOUNT_##opt)
/*
 * Maximal mount counts between two filesystem checks
 */
#define EXT2_DFL_MAX_MNT_COUNT     20    /* Allow 20 mounts */
#define EXT2_DFL_CHECKINTERVAL     0     /* Don't use interval check */

/*
 * Behaviour when detecting errors
 */
#define EXT2_ERRORS_CONTINUE       1     /* Continue execution */
#define EXT2_ERRORS_RO             2     /* Remount fs read-only */
#define EXT2_ERRORS_PANIC          3     /* Panic */
#define EXT2_ERRORS_DEFAULT        EXT2_ERRORS_CONTINUE

/*
 * Structure of the super block
 */
struct ext2_super_block {
      __u32 s_inodes_count;        /* Inodes count */
      __u32 s_blocks_count;        /* Blocks count */
      __u32 s_r_blocks_count;      /* Reserved blocks count */
      __u32 s_free_blocks_count;   /* Free blocks count */
      __u32 s_free_inodes_count;   /* Free inodes count */
      __u32 s_first_data_block;    /* First Data Block */
      __u32 s_log_block_size;      /* Block size */
      __s32 s_log_frag_size; /* Fragment size */
      __u32 s_blocks_per_group;    /* # Blocks per group */
      __u32 s_frags_per_group;     /* # Fragments per group */
      __u32 s_inodes_per_group;    /* # Inodes per group */
      __u32 s_mtime;               /* Mount time */
```

```
__u32 s_wtime;              /* Write time */
__u16 s_mnt_count;          /* Mount count */
__s16 s_max_mnt_count;      /* Maximal mount count */
__u16 s_magic;              /* Magic signature */
__u16 s_state;              /* File system state */
__u16 s_errors;             /* Behaviour when detecting errors */
__u16 s_minor_rev_level;    /* minor revision level */
__u32 s_lastcheck;          /* time of last check */
__u32 s_checkinterval;      /* max. time between checks */
__u32 s_creator_os;         /* OS */
__u32 s_rev_level;          /* Revision level */
__u16 s_def_resuid;         /* Default uid for reserved blocks */
__u16 s_def_resgid;         /* Default gid for reserved blocks */
/*
 * These fields are for EXT2_DYNAMIC_REV superblocks only.
 *
 * Note: the difference between the compatible feature set and
 * the incompatible feature set is that if there is a bit set
 * in the incompatible feature set that the kernel doesn't
 * know about, it should refuse to mount the filesystem.
 *
 * e2fsck's requirements are more strict; if it doesn't know
 * about a feature in either the compatible or incompatible
 * feature set, it must abort and not try to meddle with
 * things it doesn't understand...
 */
__u32 s_first_ino;          /* First non-reserved inode */
__u16   s_inode_size;       /* size of inode structure */
__u16 s_block_group_nr; /* block group # of this superblock */
__u32 s_feature_compat;     /* compatible feature set */
__u32 s_feature_incompat;   /* incompatible feature set */
__u32 s_feature_ro_compat;  /* readonly-compatible feature set */
__u8  s_uuid[16];           /* 128-bit uuid for volume */
char  s_volume_name[16];    /* volume name */
char  s_last_mounted[64];   /* directory where last mounted */
__u32 s_algorithm_usage_bitmap; /* For compression */
/*
* Performance hints.  Directory preallocation should only
* happen if the EXT2_COMPAT_PREALLOC flag is on.
*/
```

```
	__u8	s_prealloc_blocks;/* Nr of blocks to try to preallocate*/
	__u8	s_prealloc_dir_blocks; /* Nr to preallocate for dirs */
	__u16	s_padding1;
	__u32	s_reserved[204]; /* Padding to the end of the block */
};

#ifdef __KERNEL__
#define EXT2_SB(sb)	(&((sb)->u.ext2_sb))
#else
/* Assume that user mode programs are passing in an ext2fs superblock,
not
 * a kernel struct super_block.  This will allow us to call the feature-test
 * macros from user land. */
#define EXT2_SB(sb)	(sb)
#endif

/*
 * Codes for operating systems
 */
#define EXT2_OS_Linux		0
#define EXT2_OS_HURD		1
#define EXT2_OS_MASIX		2
#define EXT2_OS_FREEBSD		3
#define EXT2_OS_LITES		4

/*
 * Revision levels
 */
#define EXT2_GOOD_OLD_REV	0	/* The good old (original) format */
#define EXT2_DYNAMIC_REV	1	/* V2 format w/ dynamic inode sizes */

#define EXT2_CURRENT_REV	EXT2_GOOD_OLD_REV
#define EXT2_MAX_SUPP_REV	EXT2_DYNAMIC_REV

#define EXT2_GOOD_OLD_INODE_SIZE 128

/*
 * Feature set definitions
 */
```

```
#define EXT2_HAS_COMPAT_FEATURE(sb,mask)                  \
      ( EXT2_SB(sb)->s_es->s_feature_compat & cpu_to_le32(mask) )
#define EXT2_HAS_RO_COMPAT_FEATURE(sb,mask)               \
      ( EXT2_SB(sb)->s_es->s_feature_ro_compat & cpu_to_le32(mask) )
#define EXT2_HAS_INCOMPAT_FEATURE(sb,mask)                \
      ( EXT2_SB(sb)->s_es->s_feature_incompat & cpu_to_le32(mask) )
#define EXT2_SET_COMPAT_FEATURE(sb,mask)                  \
      EXT2_SB(sb)->s_es->s_feature_compat |= cpu_to_le32(mask)
#define EXT2_SET_RO_COMPAT_FEATURE(sb,mask)               \
      EXT2_SB(sb)->s_es->s_feature_ro_compat |= cpu_to_le32(mask)
#define EXT2_SET_INCOMPAT_FEATURE(sb,mask)                \
      EXT2_SB(sb)->s_es->s_feature_incompat |= cpu_to_le32(mask)
#define EXT2_CLEAR_COMPAT_FEATURE(sb,mask)                \
      EXT2_SB(sb)->s_es->s_feature_compat &= ~cpu_to_le32(mask)
#define EXT2_CLEAR_RO_COMPAT_FEATURE(sb,mask)             \
      EXT2_SB(sb)->s_es->s_feature_ro_compat &= ~cpu_to_le32(mask)
#define EXT2_CLEAR_INCOMPAT_FEATURE(sb,mask)              \
      EXT2_SB(sb)->s_es->s_feature_incompat &= ~cpu_to_le32(mask)

#define EXT2_FEATURE_COMPAT_DIR_PREALLOC       0x0001

#define EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER    0x0001
#define EXT2_FEATURE_RO_COMPAT_LARGE_FILE      0x0002
#define EXT2_FEATURE_RO_COMPAT_BTREE_DIR       0x0004

#define EXT2_FEATURE_INCOMPAT_COMPRESSION      0x0001
#define EXT2_FEATURE_INCOMPAT_FILETYPE         0x0002

#define EXT2_FEATURE_COMPAT_SUPP   0
#define EXT2_FEATURE_INCOMPAT_SUPP EXT2_FEATURE_INCOMPAT_FILETYPE
#define EXT2_FEATURE_RO_COMPAT_SUPP    (EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER| \
                                        EXT2_FEATURE_RO_COMPAT_LARGE_FILE| \
                                        EXT2_FEATURE_RO_COMPAT_BTREE_DIR)

/*
 * Default values for user and/or group using reserved blocks
 */
#define     EXT2_DEF_RESUID        0
#define     EXT2_DEF_RESGID        0
```

```
/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
      __u32 inode;                /* Inode number */
      __u16 rec_len;              /* Directory entry length */
      __u16 name_len;             /* Name length */
      char  name[EXT2_NAME_LEN];  /* File name */
};

/*
 * The new version of the directory entry.  Since EXT2 structures are
 * stored in intel byte order, and the name_len field could never be
 * bigger than 255 chars, it's safe to reclaim the extra byte for the
 * file_type field.
 */
struct ext2_dir_entry_2 {
      __u32 inode;                /* Inode number */
      __u16 rec_len;              /* Directory entry length */
      __u8  name_len;             /* Name length */
      __u8  file_type;
      char  name[EXT2_NAME_LEN];  /* File name */
};

/*
 * Ext2 directory file types.  Only the low 3 bits are used.  The
 * other bits are reserved for now.
 */
#define EXT2_FT_UNKNOWN      0
#define EXT2_FT_REG_FILE     1
#define EXT2_FT_DIR          2
#define EXT2_FT_CHRDEV       3
#define EXT2_FT_BLKDEV       4
#define EXT2_FT_FIFO         5
#define EXT2_FT_SOCK         6
#define EXT2_FT_SYMLINK      7
```

```
#define EXT2_FT_MAX         8
/*
 * EXT2_DIR_PAD defines the directory entries boundaries
 *
 * NOTE: It must be a multiple of 4
 */
#define EXT2_DIR_PAD                4
#define EXT2_DIR_ROUND              (EXT2_DIR_PAD - 1)
#define EXT2_DIR_REC_LEN(name_len) ((name_len) + 8 + EXT2_DIR_ROUND) & \
                                    ~EXT2_DIR_ROUND)

#ifdef __KERNEL__
/*
 * Function prototypes
 */

/*
 * Ok, these declarations are also in <linux/kernel.h> but none of the
 * ext2 source programs needs to include it so they are duplicated here.
 */
# define NORET_TYPE    /**/
# define ATTRIB_NORET  __attribute__((noreturn))
# define NORET_AND     noreturn,

/* acl.c */
extern int ext2_permission (struct inode *, int);

/* balloc.c */
extern int ext2_bg_has_super(struct super_block *sb, int group);
extern unsigned long ext2_bg_num_gdb(struct super_block *sb, int group);
extern int ext2_new_block (struct inode *, unsigned long,
                    __u32 *, __u32 *, int *);
extern void ext2_free_blocks (struct inode *, unsigned long,
                       unsigned long);
extern unsigned long ext2_count_free_blocks (struct super_block *);
extern void ext2_check_blocks_bitmap (struct super_block *);
extern struct ext2_group_desc * ext2_get_group_desc(struct super_block
                               * sb, unsigned int block_group, struct
                               buffer_head ** bh);
```

```
/* bitmap.c */
extern unsigned long ext2_count_free (struct buffer_head *, unsigned);

/* dir.c */
extern int ext2_check_dir_entry (const char *, struct inode *,
                        struct ext2_dir_entry_2 *, struct buffer_head *,
                        unsigned long);

/* file.c */
extern int ext2_read  (struct inode *, struct file *, char *, int);
extern int ext2_write (struct inode *, struct file *, char *, int);

/* fsync.c */
extern int ext2_sync_file (struct file *, struct dentry *, int);
extern int ext2_fsync_inode (struct inode *, int);

/* ialloc.c */
extern struct inode * ext2_new_inode (const struct inode *, int);
extern void ext2_free_inode (struct inode *);
extern unsigned long ext2_count_free_inodes (struct super_block *);
extern void ext2_check_inodes_bitmap (struct super_block *);

/* inode.c */

extern struct buffer_head * ext2_getblk (struct inode *, long, int, int *);
extern struct buffer_head * ext2_bread (struct inode *, int, int, int *);

extern void ext2_read_inode (struct inode *);
extern void ext2_write_inode (struct inode *, int);
extern void ext2_put_inode (struct inode *);
extern void ext2_delete_inode (struct inode *);
extern int ext2_sync_inode (struct inode *);
extern void ext2_discard_prealloc (struct inode *);

/* ioctl.c */
extern int ext2_ioctl (struct inode *, struct file *, unsigned int,
                       unsigned long);

/* namei.c */
```

```
extern struct inode_operations ext2_dir_inode_operations;

/* super.c */
extern void ext2_error (struct super_block *, const char *, const char *, ...)
      __attribute__ ((format (printf, 3, 4)));
extern NORET_TYPE void ext2_panic (struct super_block *, const char *,
                                   const char *, ...)
      __attribute__ ((NORET_AND format (printf, 3, 4)));
extern void ext2_warning (struct super_block *, const char *, const
                          char *, ...)
      __attribute__ ((format (printf, 3, 4)));
extern void ext2_update_dynamic_rev (struct super_block *sb);
extern void ext2_put_super (struct super_block *);
extern void ext2_write_super (struct super_block *);
extern int ext2_remount (struct super_block *, int *, char *);
extern struct super_block * ext2_read_super (struct super_block *,void *,int);
extern int ext2_statfs (struct super_block *, struct statfs *);

/* truncate.c */
extern void ext2_truncate (struct inode *);

/*
 * Inodes and files operations
 */

/* dir.c */
extern struct file_operations ext2_dir_operations;

/* file.c */
extern struct inode_operations ext2_file_inode_operations;
extern struct file_operations ext2_file_operations;

/* symlink.c */
extern struct inode_operations ext2_fast_symlink_inode_operations;

extern struct address_space_operations ext2_aops;

#endif     /* __KERNEL__ */

#endif     /* _Linux_EXT2_FS_H */
```

제８장. Linux용실행기록파일체계

실행기록 혹은 등록파일체계들은 ext2fs파일체계와 같은 보다 단순한 파일체계들의 일관성문제를 제거하였으나 속도상에서의 문제를 산생시키였다.

이제 고찰하겠지만 이 체계들은 자료기지관리체계에서처럼 파일체계안에서 갱신되는 거래라는 개념을 도입하여 파일을 갱신하기전에 갱신사건을 기억하기 위하여 가동일지를 사용한다. 일단 갱신되면 가동일지내용입구점은 실행된것으로 표기되며 이러한 보충적인 조작으로 하여 파일체계전체에 대하여서는 성능이 떨어 지게 된다. 일부 실행기록－사용등록파일체계들은 다른 체계들보다 더 빠르다.

이 장에서 우리는 JFS(실행기록파일체계)의 실현에 대하여 고찰한다. JFS는 상위블로크와 Linux가상파일체계에 의하여 요청되는 파일체계호출을 제공한다. 가동일지에 기초한 바이트준위파일체계인 JFS는 견고하고 유연한 특성을 다 가지고 있다.

주로 높은 생산성과 거래지향의 실현요구, 고성능봉사기를 목적으로 만들어 졌다는데로부터 역시 JFS는 성능과 실현성을 다같이 요구하는 클라이언트형식에 리용할수 있다.

IBM의 실행기록파일체계인 JFS는 2000년 2월에 공개되여 리용할수 있게 되였다.

JFS의 기본자료구조와 알고리듬

JFS에 대한 구체적인 연구결과는 실현상 본보기로 될만한 좋은 점들을 가지고 있다. 자료구조가 명백하며 가장 단순한 형태로 축소되여 있다. 변수이름공간이 지능적으로 리용되며 함수들도 쓸모 있게 기교화되여 있다. 이 구조와 알고리듬을 해석하여 보겠다.

상위블로크：１차집합상위블로크와 ２차집합상위블로크

상위블로크는 집합의 크기, 배정그룹의 크기, 집합블로크의 크기 등과 같은 집합범위정보를 포함한다.

2차집합상위블로크는 1차집합상위블로크의 직접 복사판이며 1차집합상위블로크가 파손된 경우에 리용된다.

이 상위블로크들은 JFS가 어떤 다른 정보에 의존함이 없이 항상 찾을수 있도록 고정된 위치에 존재한다.

상위블로크구조체는 linux\include\linux\JFS\JFS_superblock.h의 struct JFS_ super block에서 정의된다.

색인마디

JFS디스크상의 색인마디는 512byte이며 4개의 기본정보모임을 포함한다.

▼ 첫번째 모임은 JFS객체의 POSIX속성을 서술한다.

- 두번째 모임은 JFS객체의 보충적인 속성을 서술한다. 이 속성들은 VFS지원에 필요한 정보와 OS환경에 대한 특정정보 그리고 B+나무용머리부정보들을 포함한다.
- 세번째 모임은 B+나무의 뿌리의 범위배정서술자를 포함하든지 혹은 직결자료를 포함하게 된다.

▲ 네번째 모임은 확장된 속성들, 더 많은 직결자료 혹은 보충적인 범위배정서술자들을 포함한다.

디스크상의 색인마디구조체정의는 linux\include\JFS\JFS_dinode.h에서 struct dinode로 정의된다.

표준관리용편의프로그람

JFS는 파일체계를 생성하고 관리하는데 필요한 표준적인 관리편의프로그람들을 제공한다.

1. 파일체계를 생성한다. 이 편의프로그람은 정의된 구동기상에 JFS파일체계를 초기화할 때 mkfs명령의 JFS정의부분을 제공한다. 편의프로그람은 낮은 준위에서 동작하며 파일체계가 존재하게 될 기록권의 임의의 생성－초기화가 보다 높은 준위의 편의프로그람외부에서 조종된다고 가정한다. 사용자는 파일체계의 특성을 변경시킬수 있는 mkfs를 실행하여 블로크크기와 같은 정보들을 제공할수 있다.
2. 파일체계를 검사회복한다. 이 편의프로그람은 fsck명령의 JFS정의부분을 제공한다. 또한 파일체계의 일관성을 검열하고 손상된 부분을 회복하며 가동일지를 재생하고 파일체계메타자료에 대한 변경내용을 통지한다. 파일체계가 가동일지재생의 결과 깨끗하다는것이 알려 지면 다음의 동작은 더 진행되지 않는다. 파일체계가 깨끗해 진것 같지 않으면 어떤 원인에 의하여 가동일지가 완전히 그리고 정확히 재생되지 않았다는것을 지적하여 파일체계가 완전히 회복되게 한다. 완전하고도 집중적인 검사를 수행하는데서 검사－회복편의프로그람의 1차적목적은 파일체계 파손이나 고장을 막을수 있는 현실성 있는 파일체계상태를 얻는데 있다. 2차적목적은 손상에 직면하여 자료를 보존하는것이다. 이것은 편의프로그람이 파일체계 일관성을 보장하는 견지에서 자료를 대피시킨다는것을 의미한다. 편의프로그람이 구조적으로 불일치된 파일이나 등록부를 어떠한 가정도 없이 일치한 상태로 회복하는데 필요한 정보들을 가지고 있지 못할 때 자료는 손상된다. 일관성이 보장되지 않은 파일이나 등록부가 있는 경우에 전체 파일이나 등록부는 어느 한 부분도 보존하지 못하고 손상되고 만다. 손상된 등록부의 지우기에 의하여 홀로 남게 된 임의의 파일이나 등록부들은 파일체계의 뿌리에 위치하고 있는 lost+found 등록부에 배치된다. 파일체계검사－회복편의프로그람에 대하여 중요하게 고려해야 할 문제는 그것이 요구하는 가상기억의 크기이다. 대표적으로 이 편의프로그람에 요구되는 기억은 요구한 가상기억용량이 파일체계안의 개별적블로크들의 배정상태를 추적하는데 리용되기때문에 파일체계의 크기에 의존하게 된다. 파일

체계가 더 커지면 블로크수도 증가하며 따라서 이 블로크들을 추적하는데 요구되는 가상기억의 용량도 커지게 된다. JFS검사—회복편의프로그람의 설계는 파일체계의 블로크수보다도 오히려 파일체계안의 파일이나 등록부수에 의하여 가상기억요구가 제기되던 초기의 Linux fsck와는 다르다. JFS검사—회복편의프로그람의 가상기억요구는 파일과 등록부당 32byte의 차수로 되든가 혹은 파일체계크기에는 무관계하게 백만개의 파일이나 등록부들을 포함하는 체계에 대하여 약 32MB에 해당된다. 다른 모든 파일체계와 마찬가지로 JFS편의프로그람은 실제적인 파일체계에 배치된 작은 규모의 예약된 작업구역을 리용하여 블로크배정상태를 축적할것을 요구하지만 거기에 가상기억을 리용하는 방법은 피한다.

기동시 JFS의 설정

기동시에 파일체계생성편의프로그람 즉 mkfs는 올려태우기되는 매개 론리기록권(구획)안에 총체적으로 포함되는 한개의 집합을 생성한다. 이 집합은 특정한 양식에 따라 배정된 디스크블로크들의 배렬이다.

이 배렬을 다음과 같은 내용을 포함한다.

▼ JFS 집합으로서의 구획을 식별하는 상위블로크
▲ 집합안의 매 자료블로크의 배정상태를 표시하는 배정표

블로크배정표

블로크배정표는 전체 집합에 관하여 배정되였거나 비여 있는 블로크들을 추적하는데 리용된다. 집합안의 모든 파일모임은 동일한 디스크블로크의 풀을 공유하기때문에 이 배정표는 디스크블로크들을 배정하거나 배정해제할 때 집합내의 모든 파일모임*들에 의하여 리용된다.

모든 준위가 요구되지 않으면 블로크표(block map)색인마디는 매개 사용되지 않은 준위의 첫번째 페지에 구멍이 있는 성긴 파일로 될것이다.

JFS는 현실적으로 조종자료가 갱신된다는것을 확인하기 위하여 위임하는 방략을 리용한다. 《실현가능한 갱신》이라는것은 체계고장에 직면하여 일관성 있는 JFS구조체와 자원배정상태가 유지된다는것을 의미한다.

---

* 파일모임은 ext2fs파일체계와 같이 올려태우기가능한 실체들이다. 파일모임은 파일과 등록부들을 관리한다. 파일과 등록부들은 색인마디에 의하여 일관성 있게 표시된다. 매 색인마디들은 파일이나 등록부의 속성을 표현하며 디스크상의 파일이나 등록부의 자료들을 찾는데서 출발점으로 보존한다. JFS도 역시 파일모임안의 매개 색인마디의 디스크상에서 위치와 배정상태를 표현하는 표와 같은 그러한 다른 파일체계객체를 표현하는데 색인마디를 리용한다.

블로크배정표가 일관성 있는 상태에 있다는것을 확인하기 위하여 JFS는 dmap구조체에 두개의 표를 보존하는데 하나는 작업표이고 다른 하나는 영구표이다. 작업표는 현행배정상태를 기록한다. 영구표는 디스크상에서 보았거나 JFS의 가동일지안에 있는 레코드에 의하여 표시되였거나 혹은 JFS거래에 위임된 배정상태를 기록한다.

집합블로크가 개방될 때에는 영구표가 먼저 갱신되며 배정될 때에는 작업표가 먼저 갱신된다. 일부 0(령)은 빈 자원을 표시하며 1은 배정된 원천을 표시한다.

블로크배정표의 dmap조종페지들은 잎준위가 1024개의 요소를 포함한것을 제외하고는 dmap구조체의 나무와 류사한 나무를 《포함한다》.

dmap조종페지는 struct dmapct1_t에 의하여 정의되며 이 구조체는 linux\include\linux JFS\JFS_dmap.h에서 찾아 볼수 있다.

블로크배정표의 꼭대기에는 표조종구조체인 struct dbmap_t가 있다. 이 구조체에는 배정그룹의 탐색시간을 고속화하는 요약정보가 포함되여 있는데 이때 배정그룹은 (AG)평균적인 빈 공간보다 더 많은 공간을 가지게 된다.

구조체는 linux\include\linux\JFS\JFS_dmap.h에서 찾을수 있다.

블로크배정표는 정기적으로 기록되지 않는다. 이 표는 가동일지의 재등록에 의한 회복시간에 복구될수 있거나 혹은 fsck에 의하여 재구성될수 있다.

색인마디배정표

색인마디배정표는 앞방향검색문제를 해결한다.

집합과 매개 파일모임은 색인마디배정표를 보존하는데 이 표는 색인마디배정그룹(IAG)의 동적배렬이다. 이 IAG는 색인마디배정표에 필요한 자료이다. 집합에 관하여서는 색인마디배정표에 따라 넘기기된 색인마디는 집합색인마디표로 된다. 파일모임에 관하여서는 색인마디배정표에 의하여 넘기기되는 색인마디가 파일색인마디표로 된다.

매 IAG는 4KB의 크기를 가지며 디스크상에서 128개의 물리적색인마디범위를 표시한다. 매 색인마디내용은 32개의 색인마디를 포함하고 있기때문에 매개 IAG는 4096개의 색인마디를 표시하게 된다.

IAG는 집합안의 어디에나 존재할수 있다. IAG에 대한 전체 색인마디의 내용은 한개의 배정그룹에 존재한다. IAG는 모든 범위의 색인마디가 개방될 때까지 해당 AG에 속박된다. 그 시점에서 색인마디크기는 임의의 AG안에 배정될수 있으며 IAG는 그 AG에 련결된다. IAG는 linux\include\linux\JFS\JFS_imap.h파일의 iag_t구조체에 의하여 정의된다.

색인마디배정표의 첫 4K페지는 조종페지이며 이 페지는 색인마디배정표에 관한 요약정보들을 포함한다.

dinomap_t구조체에 대한 정의는 linux\include\linux\JFS\JFSmap.h파일에서 찾아 볼수 있다.

추상적으로는 색인마디배정표가 동기적으로 확장가능한 IAG구조체배렬 즉 struct iag inode_map[1..N]이며 물리적으로는 집합내에 존재하는 파일 그자체이다. 집합색인마디배정표는 집합자체마디에 의하여 서술된다.

파일모임색인마디배정표는 fileset_inode에 의하여 표현된다. 이 표의 페지는 표준B+나무첨수화에 의거하여 필요에 따라 배정되기도 하고 개방되기도 한다.

B+나무의 열쇠는 IAG페지의 바이트변위값이다.

AG자유색인마디목록

AG자유색인마디목록은 거꿀검색문제를 해결할수 있게 한다. 집합을 확장하거나 줄이는데서의 휴지시간을 줄이기 위하여 JFS는 매 집합당 허용되는 최대배정그룹들의 수를 설정할수 있다. 따라서 AG자유색인마디목록의 머리부수는 고정될수도 있다. 목록에 관한 머리부는 색인마디배정표의 조종페지안에 존재한다. *n*번째 입구점은 *n*번째 AG에 포함되는 자유색인마디를 가진 모든 색인마디배정표입구점들의 2중련결목록에 대한 머리부이다.

IAG번호는 목록안에서 첨수로 리용된다. A−1은 목록의 끝을 지적한다. 매개 IAG조종부분은 목록에 대한 앞방향 혹은 뒤방향 지적자를 포함한다. AG자유색인마디목록을 표시하는 정의는 linux\include\linux\JFS\JFS_imap.h파일의 dinomap_t구조체에 있다.

IAG자유목록

IAG자유목록은 자유색인마디번호를 탐색하는데 리용한다. 이 목록은 JFS가 임의의 대응하는 배정색인마디크기가 없이도 IAG을 찾을수 있게 한다. IAG자유목록을 표현하는 정의는 linux\include\linux\JFS\JFS_dinode.h파일의 inomap_t구조체에 있다.

파일모임배정표색인마디

집합색인마디표안의 파일모임배정표색인마디들은 특별한 형식의 색인마디이다. 이것들은 파일모임을 표현하기때문에 파일모임에 관하여서는 《상위−색인마디》로 된다. 표준색인마디자료대신에 파일모임배정색인마디는 색인마디의 웃절반에 몇가지 파일모임정의용정보들을 포함한다. 이 색인마디표들은 또한 B+나무의 파일모임배정표의 위치를 추적하는데 리용된다. 이 구조체는 linux\include\linux\JFS\JFS_dinode.h파일의 struct dinode_t구조체에 의하여 정의된다.

매 JFS객체는 색인마디에 의하여 표현된다. 색인마디는 시간표식이나 파일형식(정규적인 VS. 등록부)과 같은 객체−정의형정보를 포함한다. 이 정보들은 역시 배정크기를 기록하기 위한 B+나무를 《포함한다》.

특별히 모든 JFS메타자료구조체(상위블로크는 제외)가 《파일》로 표현된다는데 대하여 강조한다. 이 자료들에 대한 색인마디구조체를 리용하여 자료의 양식은(디스크구역에서) 계층적으로 확장가능하게 된다.

등록부는 사용자정의된 이름들을 파일이나 등록부들에 배정된 색인마디들로 넘기며 전형적인 이름계층을 형성한다.

파일은 사용자자료를 포함하며 자료에는 아무러한 제한이나 양식들이 내재되여 있지 않다. 따라서 사용자자료는 JFS에 의하여 해석되지 않는 자료의 흐름으로서 취급된다.

색인마디에 뿌리로 되여 있는 범위에 기초한 구조체는 디스크블로크들로 파일자료들을 넘기기하는데 리용된다. 《파일》은 범위순차로 배정된다.

또한 집합의 상위블로크, 디스크배정표, 파일서술자색인마디표, 색인마디, 등록부, 주소화구조체들은 JFS조종구조체 실례로 메타자료들을 이루고 있다.

다른 파일체계로부터 JFS를 분류하기 위한 설계상 특성

JFS는 현존 파일체계에 보충하는 방법으로가 아니라 시작으로부터 완전히 집중화된 실행기록수법을 가질수 있게 설계되여 있다. 게다가 JFS의 수많은 서로 다른 특성들은 JFS를 다른 파일체계들과 구별할수 있게 한다.

- *내부적JFS의 한계*
  JFS는 완전한 64bit파일체계이다. 모든 고유한 파일체계구조체마당들이 64bit로 되여 있다. 이것은 JFS가 대규모파일이나 구획을 지원할수 있게 해준다.
- *제거가능한 매체*
  JFS는 파일체계기초장치로서 플로피디스크를 지원하지 못한다.
- *파일체계의 크기*
  JFS에 의하여 지원되는 최대파일체계의 크기는 16MB이다. 파일체계의 최대크기는 파일체계메타자료구조체에 의하여 지원되는 블로크의 최대수와 파일체계블로크크기의 함수이다.
  JFS는 512TB로부터(블로크크기 512B) 4PB(블로크크기 4KB)까지의 최대파일체계를 지원할수 있다.
- *파일크기*
  최대파일크기는 VFS체계가 지원하는 최대파일크기이다. 실례로 VFS체계가 32bit만 지원하면 이것이 파일의 크기로 된다.
- *주소화구조*
  JFS는 철저하게 범위에 기초한 주소화구조로서 B+나무를 리용한다. JFS가 B+나무를 지원하는 일부 구역은 범위에 따르는 배정방법에 기초하고 있다.

파일은 B+나무의 뿌리를 포함하는 색인마디에 의하여 표현되는데 이때 B+나무의 뿌리는 사용자자료를 포함하는 범위를 서술한다. 범위는 JFS객체를 한 단위로 하여 배정된 련속집합블로크들의 렬이다. 매개의 범위는 단일한 집합내에 완전히 포함된다.

범위를 정의하는데는 두개의 값 즉 길이와 주소가 요구된다. 길이는 집합의 블로크크기를 단위로 하여 계산된다. JFS는 범위를 표시하는 24bit값을 사용한다. 그리하여 하나의 범위는 1부터 ($2^{24}-1$)의 집합블로크단위로 배렬될수 있다. 주소는 집합블로크들의 단위(집합의 시작으로부터 블로크의 변위)로 된 범위의 첫번째 블로크의 주소이다. 범위는 다중배정그룹들로 늘쿨수 있다. 이 범위들은 새로운 범위를 삽입하거나 특정한 범위를 찾아 내는 등의 성능최량화를 위하여 B+나무로 첨수화된다. 512byte의 집합블로크범위(허용가능한 최소값)는 최대로 $512*(2^{24}-1)$까지 정의할수 있다(8GB보다 적게). 4096바이트의 집합블로크범위(허용가능한 최대값)로써는 최대크기를 $4096*(2^{24}-1)$byte까지 정의할수 있다(64GB보다 적게). 이 한계는 단일범위의 경우에만 적용한다. 이것들은 파일체계전반에 걸쳐 그 어떤 영향도 주지 않는다.

사용자정의집합블로크와 결합된 범위에 기초한 파일체계는 JFS로 하여금 내부의 토막화를 따로따로 지원할수 없게 한다. 사용자는 작은 범위로 된 많은 파일들로 집합의

내부토막화를 최소화하기 위하여 작은 규모의 집합블로크범위(실례로 512byte)로서 집합을 배치할수 있다.

일반적으로 JFS의 배정방법은 최소린접배정방법으로 진행하여 즉 가능한 정도의 크기로 린접되도록 배정함으로써 련속적인 배정이 최대화되도록 한다. 이 방법은 대규모의 I/O전송에서도 성능을 개선한다.

이름에 의하여 정렬된 B+나무는 특정한 등록부의 입구점을 지적하는 능력을 높이는데 리용한다.

JFS는 적극적인 블로크배정조작에 따라 파일안의 론리적변위값들을 디스크상의 물리적주소로 넘기는데서 집약적이고도 효과적이고 유연한 구조체로 된다. 이미 언급된바와 같이 범위는 한개 단위로 파일안에 배정된 련속적인 블로크들의 렬이며 론리적변위/길이/물리적주소를 포함하는 3중구조에 의하여 서술된다. 주소화된 구조는 범위서술자에 의하여 3중다중화되고 색인마디에 뿌리로 보존되며 파일안의 론리적편위에 의하여 열쇠화된 B+나무이다.

JFS에서 B+나무의 광범한 리용

이 부분에서는 파일의 륜곽적구조에 리용되는 B+나무구조를 서술한다. JFS에서 진행되여야 할 가장 공통적인 조작들 즉 범위의 읽기, 쓰기성능을 높이기 위하여 B+나무를 선택한다. B+나무는 또한 파일의 범위를 효과적으로 첨가 혹은 삽입할수 있는 기능도 제공한다.

좀 일반적이기는 하지만 JFS에서는 B+나무에 리용된 블로크들뿐아니라 파일자료를 제거하였는가를 확인하기 위하여 파일제거시 전체 B+나무를 조사해야 할 필요가 제기된다. B+나무는 또한 추적에도 아주 효과적이다. 범위배정서술자(xad 구조체)는 범위를 서술하며 파일을 표시하는데 요구되는 두개이상의 마당들을 첨가한다. 이러한 마당으로는 범위를 표현하는 론리적바이트주소를 서술하는 변위와 기발마당을 들수 있다. 범위배정서술자구조체는 linux\include\JFS\JFS_xtree.h에 struct xad로 정의된다.

여기에는 등록부를 제외하고 JFS의 모든 첨수객체들을 위한 한가지 일반적B+나무첨수구조체가 있다. 첨수화되는 자료는 객체에 의존하며 B+나무는 나무에 의하여 서술되는 xad자료의 변위에 의하여 열쇠화된다. 입구점들은 xad구조체의 변위에 따라 정렬된다. xad구조체는 B+나무의 한개 색인마디에 있는 입구점이다.

일단 색인마디에 8개의 xad구조체가 채워 지게 되면 보다 많은 구조체에 관하여 색인마디의 마지막《상한》을 리용하려는 시도도 있다. 만일 색인마디의 di_mode마당에 INLINEEAbit가 설정되면 색인마디의 마지막상한을 리용할수 있다. 만일 색인마디의 리용가능한 모든 xad구조체가 다 리용되면 B+나무는 분리되여야 한다.

디스크색인마디의 두번째 부분의 아래쪽은 색인마디의 두번째 절반부분에 무엇이 기억되여 있는가를 알려 주는 자료서술자가 포함되여 있다. 두번째 절반부분이 충분히 작다면 파일에 대한 직결자료를 포함할수 있다. 만일 파일자료가 색인마디에 관한 직결자료에 정합되지 않으면 범위에 포함되며 색인마디는 B+나무의 뿌리정점을 포함하게 된다. 머리부는 몇개의 xad가 사용중에 있으며 몇개가 리용가능한가를 지적한다.

일반적으로 색인마디는 B+나무의 뿌리에 대하여 8개의 xad를 포함한다. 만일 파일에 대하여 8개나 그보다 적은 범위가 있으면 이 xad구조체들은 B+나무의 잎마디로 되며 범위를 표현한다. 그렇지 않으면 색인마디의 xad구조체들은 잎들을 지적하든지 혹은 B+나무의 내부마디를 지적하게 된다.

잎 마 디

JFS는 B+나무의 잎마디에 4KB의 디스크공간을 배정한다. 잎마디는 머리부를 가지는 xad입구점들의 배렬이다. 머리부는 마디의 첫번째 빈 xad입구점을 지적하며 배정되지 않은 입구점다음의 모든 xad입구점들을 제거한다. 8개의 xad입구점들은 색인마디로부터 입구점에 복사되며 머리부는 첫번째 빈입구점으로서 아홉번째 입구점을 지적하기 위하여 초기화된다. 다음 B+나무의 뿌리를 색인마디의 첫번째 xad구조체로 갱신하는데 이 구조체는 새로 배정된 잎마디를 지적한다.

이 새로운 xad구조체에 대한 변위는 잎마디의 첫번째 입구점의 변위로 된다. 색인마디안의 머리부는 현재 하나의 xad만 사용되고 있다는것을 지적하기 위하여 갱신되게 된다. 머리부도 또한 색인마디가 현재 순수 B+나무뿌리를 포함하고 있다는것을 지적하기 위하여 갱신되여야 한다.

새 범위들이 파일들에 보충되므로 마디가 채워 질 때까지 순서대로 같은 잎마디에 계속 보충되게 된다. 이런 견지에서 4KB의 새로운 디스크공간이 다른 잎마디에 의하여 배정되며 그 색인마디로부터 두번째 xad구조체가 새롭게 배정된 이 색인마디를 지적할수 있게 설정된다. 이러한 과정은 8개의 구조체가 색인마디에 다 채워 질 때까지 계속 진행되며 이때 B+나무의 다른 뿌리부분이 생성된다. 이 갈라 진 부분은 순수 나무의 탐색을 추적하는데만 리용되는 B+나무의 내부마디들을 생성하게 된다.

내 부 마 디

JFS는 B+나무의 내부마디를 위하여 4KB의 디스크공간을 배정한다. 내부마디는 잎마디와 꼭 같이 볼수 있다. 잎마디를 가지고 있기때문에 8개의 xad입구점들은 그 마디로부터 내부마디로 복사되며 머리부는 9번째 입구점을 지적할수 있게 초기화된다.

다음 JFS는 색인마디의 첫 xad구조체가 새로 배정된 내부색인마디를 지적할수 있게 함으로써 B+나무의 뿌리를 갱신한다. 색인마디의 머리부는 B+나무에 오직 한개의 xad가 리용되고 있다는것을 지적할수 있도록 갱신된다. linux\include\linux\JFS\JFS_xtree.h파일은 구조체 struct xtpage_t의 B+나무뿌리에 대한 머리부를 표시한다. linux\include\linux\JFS\JFS_btree.h파일은 구조체 struct btpage_t에 있는 잎마디든가 혹은 내부마디를 위한 머리부이다.

가변블로크크기

JFS는 매 파일체계당 512, 1024, 2048, 4096byte의 블로크크기를 지원한다. 이것은 사용자가 응용프로그람실행환경에 기초하여 공간리용을 최량화할수 있게 한다. 블로크크기를 작게

하면 등록부나 파일안에 내부토막의 수가 작아 지므로 효과적인 공간적성능이 얻어 진다. 그러나 작은 블로크들을 리용하면 큰 블로크크기를 리용할 때보다 블로크배정동작이 더 자주 진행되기때문에 경로의 길이가 길어 지게 된다. 봉사기에서는 공간적리용보다 오히려 성능상의 문제가 선차적인 고려대상으로 되기때문에 기정블로크크기는 4096byte로 제한한다.

JFS는 공간이 더 이상 필요하지 않을 때에는 그것을 개방하여 요구되는것만큼 디스크색인마디공간을 배정한다. 이 방법은 파일체계생성시에 고정된 디스크색인마디공간을 예견하지 않아도 되게 하며 따라서 파일체계가 포함하는 파일과 등록부의 최대수를 평가하지 않아도 된다. 또한 이 방법은 고정된 디스크위치로부터 디스크색인마디들을 분리시킨다.

등록부조직

JFS에는 두개의 서로 다른 등록부조직방법이 있다.

첫번째 조직방법은 작은 등록부들에 리용하는데 등록부의 내용을 등록부의 색인마디에 기억한다. 이 방법은 등록부블로크 I/O를 분리시킬 필요성뿐아니라 기억을 분리하여 배정할 필요성도 느끼지 않게 한다. 8개까지의 입구점들이 색인마디안에 즉시삽입방법으로 기억될수 있으며 자기자체(.)와 상위(..)를 제외하고는 따로따로 된 구역에 기억된다. 두번째 조직방법은 보다 큰 등록부들에 리용되며 매 등록부는 이름에 관하여 열쇠화된 B+나무로써 표현된다. UNIX등록부와 마찬가지로 Linux등록부는 ls명령에 스위치 ”−i” 를 리용하여 찾아 볼수 있다. 이 조작은 전통적인 비정렬등록부조직방법과 비교할 때 등록부고속검색, 삽입, 삭제능력을 제공한다.

성긴파일과 밀집파일에 대한 JFS의 지원

JFS는 매 파일체계에 기초하여 성긴파일*과 밀집파일을 지원한다. 성긴파일은 파일블로크가 장애로 인하여 씌여 지지 않았다는것을 먼저 검증하지 않고도 자료를 임의의 위치에 써넣을수 있게 한다. 알려 진 파일크기는 리용한 바이트수로는 제일 높지만 파일에 주어 진 임의의 블로크의 실지적배정은 쓰기조작이 그 블로크우에서 완성될 때까지 발생하지 않는다. 실례로 성긴파일로 설계된 어떤 파일체계에 새로운 파일이 생성되였다고 하자. 응용프로그람은 파일안의 100개의 블로크에 한개 자료블로크를 쓴다. JFS는 디스크공간상의 한개 블로크가 파일에 배정되였지만 파일의 크기가 100개 블로크라고 통보한다. 만일 응용프로그람이 파일의 블로크50을 읽는다면 JFS는 값이 모두 0인 한개의 블로크를 귀환시킬것이다. 다음 응용프로그람이 파일의 블로크50에 한개의 자료블로크를 쓴다고 가정하자. JFS는 역시 이 파일의 크기를 여전히 100개라고 통보할것이며 이때 디스크공간의 두개 블로크가 파일에 배정된다. 성긴파일은 대규모의 론리공간을 요구하지만 이 공간의 부분모임만을 리용하는 응용프로그람에서는 흥미 있는 문제로 된다.

* 모든 첨수들은 제일 좋은 대기렬성능을 얻기 위하여 기억안에 보존된다. 이 경우에 S.B우의 성긴첨수들은 모든 열쇠들이 그 성긴첨수안에 나타나지 않기때문에 밀집첨수보다는 좋지 못하다. 밀집파일에서는 디스크자원들이 파일의 크기를 포함할수 있게 배정된다.

앞의 실례에서 첫번째 쓰기에서는 파일에 배정될 디스크공간의 블로크수가 100으로 얻어 질것이다. 암시적으로 씌여 진 어떤 블로크상에서의 읽기조작은 성긴파일의 경우와 꼭 같이 모든 값이 0인 블로크를 귀환시킬것이다.

집합과 파일모임

JFS의 용어중에는 집합과 파일모임이라는 표현이 있는데 이 절에서는 이 용어들에 대하여 고찰하게 된다.

파 일

파일은 B+나무의 뿌리를 포함하는 색인마디에 의하여 표현된다. 이때 B+나무는 사용자자료를 포함하는 범위를 서술한다. B+나무는 범위의 변위에 의하여 첨수화된다.

등 록 부

JFS등록부에는 가동일지화된 메타자료파일이 있다.

UNIX등록부와 같이 Linux등록부는 곧 파일의 잎이름과 색인마디번호사이의 관계이다. 파일의 색인마디번호는 ls명령에 ”-i”스위치를 리용하여 찾아 볼수 있다.

등록부는 포함된 객체들을 지적하는 등록부입구점들로 구성된다.

등록부입구점은 이름을 색인마디번호와 련결하며 정의된 색인마디는 정의된 이름을 가진 객체를 표현한다. 특정한 등록부입구점을 지적하기 위한 능력을 제공하기 위하여 이름에 의하여 정렬된 B+나무가 리용된다.

등록부색인마디의 di_size마당은 바로 등록부B+나무의 잎페지들을 표현한다. 등록부의 잎마디들이 색인마디안에 포함될 때 di_size의 값은 256으로 된다. 등록부는 자기자신 (”.”)과 상위 (”..”)들에 대한 특정한 입구점들을 포함하지 않는다. 대신 이것들은 색인마디자체에 표현된다. 자기자신이라는것은 등록부자체 색인마디번호를 말하며 상위라는것은 색인마디에 특별한 마당 즉 linux\ include\linux\JFS\JFS_dtree.h파일의 idotdot, struct dtroot_t을 말한다.

등록부색인마디는 일반정규파일과 류사한 방식의 B+나무뿌리를 포함하며 이름에 의하여 열쇠화된다. 등록부B+나무의 잎마디는 등록부입구점을 포함하며 완전한 입구점이름으로 열쇠화된다.

등록부B+나무는 B+나무의 마지막내부마디에 관하여 뒤붙이압축을 리용한다. 나머지 내부마디들은 동일한 뒤붙이압축을 리용한다. 뒤붙이압축은 선행입구점으로부터 현행입구점을 구분할만큼 충분한 문자수로 이름을 제한한다.

가 동 일 지

JFS가동일지는 매개 집합안에 보존되며 메타자료조작에 대한 정보를 기록하는데 리용된다. 가동일지는 파일체계생성편의프로그람에 의하여 설정되는 양식을 가진다. 단일가동일지는 집합안의 다중올려태우기된 파일모임에 의하여 동시에 리용될수 있다. 가동일지에 기초한 몇가지 회복측면은 아주 흥미 있다.

첫째로 JFS는 오직 메타자료에 관한 조작만을 등록한다.

그리하여 가동일지만을 재현시키는 방법으로 파일체계안의 구조적관계의 일관성과 자원배정상태를 회복한다. 이 방법은 파일자료는 가동일지에 기록하지 않으며 일관성상태에 대한 자료만 회복한다. 동시에 일부자료파일들은 잃어 질수도 있고 회복후에 정상이 아닐수도 있으며 따라서 자료의 일관성에 대한 요구가 강한 사용자들은 동기식I/O를 리용해야 한다.

가동일지기능은 매체에 오유가 발생할 때에는 특별한 효과가 없다. 특히 디스크에 가동일지나 메타자료를 쓰는 과정에 생기는 I/O오유는 시간소비를 없애고 체계의 폭주후에 파일체계를 정상상태로 회복하기 위하여 잠재적인 장애를 집중적으로 완전히 검사하여야 한다. 이것은 불량블로크재배치가 어떤 기억관리기나 JFS의 토대장치들의 기본특성으로 된다.

JFS가동일지기능은 메타자료에 대한 변경(실례로 unlink())을 포함한 파일체계의 조작이 귀환코드를 성과적으로 돌려 줄 때 조작효과성이 파일체계에 통보되며 설사 체계가 폭주된다고 해도 알려 지게 하는것이다. 실례로 파일이 일단 성공적으로 제거되면 그 파일은 체계가 폭주되여 재시동할 때 다시 나타나지 않게 된다.

이러한 가동일지등록형식은 매개 색인마디나 메타자료를 변경하는 VFS에 가동일지디스크에 대한 동기적쓰기를 도입한다(자료기지 숙련자에게 있어서 이 조작은 비절취완충기방법을 리용하는 반복작업, 물리적잔상, 미리쓰기가동일지등록규약 등이다.). 성능상견지에서 보면 이 조작은 일관성을 위하여 다중동기메타자료쓰기에 의존하는 많은 비실행기록파일체계와 비교되는 정도이다. 그러나 다른 실행기록파일체계 즉 Veritas VxFS와 Transarc Episode와 같은 실행기록파일계에 비해 보면 성능상 부족점이 있다. 이 체계들은 서로 다른 가동일지기능형식을 리용하며 가동일지자료를 디스크상에 서서히 기록한다. 다중병렬조작을 수행하는 봉사기환경에서 이 성능비는 다중동기쓰기조작을 단일쓰기조작과 결합하는 그룹위임에 의하여 감소된다. JFS의 가동일지등록형식은 상당히 개선되였으며 현재는 비동기적가동일지등록을 제공하는데 이것은 파일체계의 성능을 제고한다.

파일구조와 접근조종

만일 어떤 방법으로 파일을 유연성 있게 조종하여 여러 사용자들에게 공유시키려면 여러가지 안전한 방식들이 요구된다.

이 요구는 다음과 같은 내용들을 포함한다.

▼ 다른 사람으로 가장한 침입자로부터 보호하는것
- 특별하게 접근이 허용된 사람들에 의한 사고나 고의적인 행동으로부터 보호하는것
- 특별히 접근이 금지된 사람의 고의적인 행동이나 파괴행위로부터 보호하는것
- 자기자신에 의하여 초래된 파괴로부터 보호하는것
- 필요하다면 한명의 사용자 혹은 사용자그룹에 의하여서만 접근이 허용되는 완전비공개성
- 장치 혹은 체계프로그람의 오유나 고장으로부터의 보호
- 인증되지 않은 사용자에 의한 간섭으로부터 체계의 안전성을 자체로 보호하는것

▲ 다른 보호수단들의 지나친 응용으로부터 보호하는것

다음 장들에서 Linux의 일반개념으로서 보호문제가 어떻게 실현되며 개별적파일체계들에 어떻게 적용되는가에 대하여 서술한다.

제9장. Linux용Reiser파일체계

가장 오랜 Linux용실행기록파일체계들중의 하나는 ReiserFS(Reiser파일체계)이다. 이 대상과제는 한스 Reiser(Hans Reiser)에 의하여 개발되였다.

ReiserFS는 설계와 프로그람구성에 있어서 우수한것이지만 다른 비Linux구성요소들과의 련관성이 부족한 결함도 가지고 있다.

ReiserFS의 기본목적은 폭주후에 회복시간을 최대로 줄이며 업무처리에서 파일체계의 메타자료를 갱신할수 있게 하는것이다. ReiserFS는 고속이며 특히 소규모파일과 많은 파일을 포함하고 있는 등록부에 대해서도 역시 속도가 빠르다.

앞으로의 목표는 전원투입시 보호목록을 가지는 원래의 파일체계만큼 빨리 예비쓰기가동일지등록을 수행하게 하는것이다.

이 장에서는 ReiserFS설계와 개념의 구체적측면뿐만아니라 설치와 관리 등과 같은 보다 실천적인 문제들을 취급한다.

파일체계의 이름공간

ReiserFS의 중심적개념들중의 하나는 유일화된 이름공간이다. 한스 레이져는 《파일》과 등록부의 두 객체를 다 포함하는 파일체계를 만들려고 하였다.

실례를 들어 설명하기 위하여 bmoshe는 사용자이고 bmoshe/mail은 우편통이며 bmoshe/mail/Message-ID/20000615091245.A2500@moshebar.com 은 전자우편이며 bmoshe/mail/Message-ID/20000615091245.A2500@moshebar.com/t0은 T0이라고 하겠다.즉 해당한 e-mail(전자우편)의 머리부마당이다. 그룹들과 결합하여 전화상에서 Avivit의 모든 e-mail을 찾기 위하여 mail/[form/avivit phones]를 검색할수 있어야 한다. 이러한 유일화되고 닫겨 진 이름공간은 프로그람작성 특히 객체지향프로그람작성을 쉽게 하는데서 커다란 영향력을 가진다. 다음의 명령에서 알수 있는바와 같이 매개객체는 그것의 마당접근자와 부분이름과 같은 어음변화조작을 가지는 그자체의 이름짓기문맥으로 자연스럽게 생각할수 있다. 우의 실례를 의역하면 다음과 같이 된다.

```
moshe.getMailBox().getMessageByID("20000615091245@moshebar.com")
```

플랜 9(Plan 9)와 인퍼노(Inferno)조작체계들과 같이 잘 알려 진 체계들은 분명히 류사성을 가진다.한스 레이져가 제기한바와 같이 유일한 이름공간은 하나도 없지만 "everything-is-a file"이라는 개념은 좀 생각해 볼 필요가 있다. 하지만 그가 제기한 파일체계에 대한 추상화는 여전히 개념상 몇가지 의문스러운 점들도 가지고 있다. 무엇보다도 속성의존문제는 속성이 객체가 아니라는데로부터 성립하지 않는다.

사실상 속성은 객체의 마당이 아니다(설사 속성이 그러한 마당들을 드러내 보일수도 있지만). Reiser의 그룹화와 순서화의 통일문제는 아주 어려운 문제라는것을 알수 있게

한다. 레이져가 제기한것처럼 초의미적문법은 가능한 모든 요구에 대하여 관리가능한 몇가지 렬을 선정하는것이다. 일반화, 집합, 분류 그리고 관계는 is−a, has−property, is−an−instance−of, is−a−member−of렬에 각각 대응한다. 레이져가 제기한 문제는 속성을 알려면 그 관계를 알아야 한다는것이다. 레이져의 실례를 리용하여서는 추진−공급(propulsion−provider−for)관계를 우에서 설명한 정규관계로 어떻게 분해하는가를 모르는 이상 속성을 알아 낼수 없다.

이로부터 이름공간의 통일과 닫김에 대하여서는 더 연구하여야 한다. 생성체계에 관한 ReiserFS를 리용할수 있으며 또 효과적이라는것을 알고 있지만 아직도 proof−of−concept파일체계와 research−and−development도구를 리용하고 있는데 이것들은 그 어떤 의미에 의하여 생성된것은 아니다.

이제 ReiserFS의 몇가지 기술적설계개념을 더 연구해 보자.

파일경계의 블로크정돈

ReiserFS는 디스크상에서 블로크들로 파일경계들을 정돈한다. 이것은 몇가지 리유로부터 설명할수 있다. 즉 블로크의 수를 최소화하기 위하여 파일의 량단을 넓힌다(이것은 파일의 량끝점위치가 불투명할 때 다중블로크파일들에 대하여 특별히 효과적이다.). 이것은 디스크나 완충기에 완전히 채워 지지 않은 블로크들을 기억시킴으로써 있을수 있는 랑비를 피하는것, 기준장소가 정해 졌을 때 역시 다 채워 지지 않은 블로크에 대하여 개개의 접근으로 대역폭의 랑비를 없애는것, 등록부안의 매 파일에 접근하기 위하여 요구되는 평균선행불러내기블로크수를 줄이는것 등인데 보다 간단한 코드로 실현할수 있다.

보다 간단한 파일체계의 블로크정돈코드는 디스크조종장치의 단위를 구분하기 위하여 계층화할 때, 공간배정으로부터 알고리듬을 완충화할 필요가 없을 때 또한 균형나무알고리듬에서 고찰한바와 같이 마디들의 묶음을 최량화하지 않아도 될 때 필요하다.

한스 레이져는 디스크공간의 랑비를 피하기 위하여 처음부터 작은 파일들을 집합화하려고 했다. 제일 간단한 해결방법은 등록부안의 모든 파일들을 한개 파일이나 혹은 한개 등록부에 집합시키는것이였다.

하지만 한개 파일이나 등록부로 집합시키는 문제는 집합의 마지막블로크부분을 랑비한다. 한개 등록부안에 몇개의 작은 파일들이 있으면 그것들을 등록부의 상위에 어떻게 집합시키겠는가 하는것이다. 처음에 한개 등록부에 몇개의 작은 파일만이 있고 다음에 많은 파일들이 있으면 OS는 그것들을 집합시키는데 어느 준위를 리용하겠는가에 대하여 결심하며 어느 때 그것들을 상위등록부로 돌려 보내며 등록부에 직접 기억시키겠는가?

물론 이 문제는 균형나무에서 마디의 균형화와 밀접한 관계가 있다. 균형나무방법을 리용하면 동적으로 집합된 순서화된 파일들을 보다 낮은 준위의 마디로 리용하므로 정적집합이나 그룹화에서 제기된 질문들을 피할수 있다.

ReiserFS방법에서 파일이나 등록부는 다같이 균형나무에 기억된다. 균형나무는 작은 파일, 등록부입구점, 색인마디 그리고 큰 파일의 마지막끄리부분과 함께 매개 요소들이 블로크정돈에 대한 요구를 낮추게 함으로써 더 효과적으로 묶어 질수 있게 하며 색인마디들에 대하여 고정된 공간배치의 리용을 없앨수 있게 한다.

균형알고리듬에 의하여 큰 파일들의 본체는 나무에 첨가한 양식화되지 않은 마디들에 기억되지만 효과적이며 가능한 이동기능은 떨어 진다. NTFS와 XFS도 집합파일은 쓰지 않는다. 이 체계들은 정합될만큼 충분히 작으면 정적으로 배정된 색인마디의 블로크주소마당에 적은 파일들을 기억시키는 한이 있더라도 블로크들을 파일별로 정돈시킨다.

모두가 서로 다른 립도를 가지는 의미론(파일), 묶음화(블로크/마디), 고속완충기억화(선행읽기크기)디스크들의 장치적대면부와 페지화는 서로 련관된 론제로 제기된다. 흔히 이와 같이 립도가 서로 다르다는것을 리해하고 그것들을 한개 층이 다른 층들에 강하게 영향을 주지 않는 개개의 층으로 추상화함으로써 공간/시간성능을 개선해 나갈수 있다. ReiserFS는 의미론적계층이 비립도화된 순서화를 파일경계에 의하여 립도화된 계층으로 이전시키는데서 일정하게 전진하였다.

ReiserFS의 코드알고리듬을 읽으면서 ReiserFS를 더 전진시키는데 필요한 령역들을 리해하는 문제가 제기된다.

균형나무와 대규모파일 I/O

I/O효과성과 대규모파일들의 블로크크기사이의 호상작용을 리해하는것은 아주 복잡한 문제이며 이때 공간은 전통적인 체계고찰방법으로는 취급할수 없다.

ReiserFS에는 공간기억을 위한 재묶음휴지시간이 초래되기때문에 블로크를 크게 하여야 하는 구성방식상의 취약성이 동반된다.

▼ 파일의 끄리(4KB보다 작은 파일들은 모두 끄리가 있다.)가 전체 색인마디를 차지할만큼 커지게 할 때 양식화된 마디로부터 비양식화된 마디로 변환된다.
- 한개 마디보다 더 작은 끄리는 두 마디사이에 퍼질수 있으며 이때 두개 마디의 기준위치가 불량하면 보다 많은 I/O를 요구하게 된다.
- 한개 마디로 다중끄리들을 집합하는것은 끄리로부터 파일의 본체를 분리시키는 결과를 초래하며 이때 읽기성능이 떨어 진다. 크기상으로 마디와 가까운 ReiserFS에서는 영향이 현저하게 나타난다.

▲ 양식화된 마디의 마지막항목이 아닌 파일이나 끄리에 한개 바이트를 추가할 때 평균적으로 전체 마디의 절반이 기억에 옮겨 진다.

어떤 응용프로그람이 몇개의 작은 완충화된 쓰기방식을 안정시키는 방법으로 I/O를 수행할 때 ReiserFS는 I/O를 완충시키지 않게 하는데 많은 비용이 든다.

실재하는 파일체계적재를 실현하는 대다수의 응용프로그람들은 표준C언어서고의 I/O

함수들을 리용하여 I/O완충기화를 아주 단순하고도 효과적으로 실현하고 있다.

작은 블로크/범위에 대한 접근을 피하기 위하여 ReiserFS는 I/O의 효과성을 개선하고 있다. VFS와 같이 크기에 기초한 파일체계와 ext2fs와 같이 쓰기-클라스터형파일체계들은 기정값으로 주어 진 1K블로크보다 512byte블로크를 선택하여 리용할 때 효과성이 높지 못하다. ext2fs는 1K를 리용할 때보다 4K를 리용할 때 28%정도 속도가 높다.

하지만 ext2fs의 개발자들은 공간랑비를 없애기 위하여 1K블로크를 리용할것을 요구한다.

ext2fs나 ReiserFS에 보충할수 있는 가치 있는 대규모파일최량화방법이 있다. 이러한 견지에서 두 파일체계는 어느것이든지 원시적이며 그중에서도 ReiserFS가 좀 더 원시적이라고 볼수 있다.

ReiserFS에는 블로크크기를 증가시키는 방법외에 대규모파일에 대한 가치 있는 최량화수법이 없다.

블로크크기를 제외하고는 다음의것들사이에 본질적인 차이가 없다.

▼ 나무에 마디쓰기를 보충한 비양식화된 마디에 대한 지적자를 첨가하는 비용
▲ 색인마디의 블로크쓰기를 보충한 주소마당의 첨가

블로크크기를 제외하고 고성능대규모파일체계의 1차적결정항목은 중소규모의 균형나무를 파일에 리용하겠는가를 결정하는 항목과 같다.

대규모파일들에 대해서는 3중간접블로크를 지적하는 색인마디에 의하여 형성되는 나무보다 더 균형화된 나무를 가지지 않는다는 우점이 있다. 소규모파일에서는 균형마디들의 기억대역비용으로 인하여 성능상 휴지시간이 존재한다.

직렬화와 일관성

최소직렬화와 자료치환으로 회복기능을 보장하는 문제는 불가피하게 고성능설계문제를 특징 짓게 된다. 이제 직렬화에서 나서는 두가지 극단적인 문제를 정의하여 이러한 근거가 명백해 지게 할수 있다. 모임의 매 블로크요청이 핵심부의 승강기알고리듬*과 선행요청의 완성을 기다리는 디스크구동기펌웨어(firmware)에 직렬로 주어 지는 I/O모임의 상대적속도를 고찰하자. 이제 모든 블로크요청이 승강기알고리듬에 주어 지고 다음 그것이 모두 정돈되며 정돈된 순서로 수행되도록 하는 한가지 극단을 고찰하자. 직렬화되지 않은 한가지 극단에서는 순환과 찾기에 드는 비용으로 크기를 더 빨리 순서화할수 있다. 불필요하게 직렬화된 I/O는 승강기알고리듬이 모든 I/O의 배치작업을 넌대순으로 보다도 배치순으로 할수 없게 한다. 대다수의 고성능설계들에서는 I/O조작이 디스크상에서의 구조적배치순서와 발표된 순서로 진행되게 하는데로 집중되고 있다.

ReiserFS는 회복성을 보장하기 위하여 보존목록(preservelist)이라는 새로운 도식을 받아 들이고 있는데 이것은 새로운 메타자료의 쓰기에 의하여 변경된 메타자료의 중복쓰기를 피하게 한다.

---

* 승강기알고리듬은 디스크에 대한 I/O조작을 계획화하는 핵심부안의 프로그람이다.

나무의 정의

균형나무를 응용프로그람에 의하여 정의된 열쇠들의 모임이라고 하면 나무의 설계목적은 이 열쇠에 의한 탐색을 최량화하는것이다. ReiserFS에서 나무의 목적은 참조위치와 공간적이며 효과적인 객체의 묶기를 최량화하는것이며 이때 열쇠는 알고리듬에 따라 정의된다. 열쇠는 파일체계안의 색인마디번호대신에 리용되며 따라서 색인마디번호를 파일위치로 넘기기하는것보다 오히려 열쇠의 마디위치(내부마디)로 넘기기를 한다. 열쇠는 색인마디번호보다 길지만 하나이상의 파일이 한 마디에 기억될 때는 고속완충되는데 필요한 열쇠길이가 오히려 더 짧다.

ReiserFS나무는 파일이름이 어느 한 순간에 한개의 성분을 해결할것을 요구한다. 이 요구는 이름이 필요한지 혹은 최량적인지를 앞으로 더 연구하는데서 하나의 문제점으로 된다. 이 문제는 그것을 실현하는것보다 더 복잡한 론의점이다. 어느 한 순간에 등록부검색은 한가지 압축형식을 완성하게 되는데 이것은 다른 이름공간의 올려태우기와 체계확장자, 안정성 그리고 보다 확장강화된 의미론적문법을 더 단순하게 한다. 파일을 작게 취하면 등록부가 커지기때문에 나무알고리듬을 리용하는 이 등록부기술이 대규모적인 등록부에 대하여 대다수 다른 파일체계들보다 훨씬 더 효과적인것으로 되였다. 나무는 3개의 마디형식을 가진다. 하나는 내부마디이고 다른 하나는 양식화된 마디이며 또 다른 하나는 양식화되지 않은 마디이다. 내부마디와 양식화된 마디는 열쇠의 순서로 정렬된다(양식화되지 않은 마디는 열쇠를 포함하지 않는다.).

내부마디는 경계열쇠에 의하여 분리되는 부분나무에 대한 지적자를 포함한다. 부분나무에 대한 지적자보다 앞서 있는 열쇠는 그 부분나무의 첫번째 양식화된 마디에 있는 첫 중복열쇠이다. 내부마디는 양식화된 마디가 열쇠에 대응하는 항목을 포함하는가를 결정하는데 쓰인다.

ReiserFS는 뿌리마디로부터 출발하여 내용을 검열하며 부분나무가 요구된 열쇠에 대응하는 항목을 포함하는가를 결정할수 있다. 뿌리마디로부터 ReiserFS는 매개 마디에서 갈라 지면서 요구되는 항목을 포함하는 양식화된 마디에 도달할 때까지 나무를 따라 아래로 내려 온다.

나무의 첫번째(바닥) 준위는 양식화되지 않은 마디를 포함하며 두번째 준위는 양식화된 마디를 포함하며 그우의 모든 준위는 내부마디를 포함한다. 제일 높은 준위는 뿌리마디를 포함한다. 준위수는 나무의 꼭대기에 새로운 뿌리마디를 요구되는만큼 첨가함에 따라 증가하게 된다.

나무의 뿌리로부터 양식화된 모든 준위까지의 모든 경로는 길이가 같다. 양식화되지 않은 준위까지의 경로도 역시 같지만 양식화된 준위까지의 경로보다는 한마디가 더 길다. 경로길이에서의 이러한 등가성과 그것을 제공하는 고속전개는 고성능을 얻는데서 핵심기능이다.

양식화된 마디들은 네가지 형식 즉 직접, 간접, 등록부, 상태자료로 된 항목을 포함한다. 모든 항목들은 항목을 찾거나 정렬시키는데 사용되는 유일한 열쇠를 포함한다. 직접항목은 파일의 꼬리를 포함하는데 이 꼬리는 파일의 마지막부분이다. 간접항목은 양식화되지 않은 마디에 대한 지적자를 포함한다. 모든 파일의 꼬리는 양식화되지 않은 마디

안에 포함된다. 직접항목은 그 항목안에 첫번째 등록부의 입구점열쇠를 포함하며 그뒤에는 많은 등록부의 입구점들이 놓인다.

파일은 간접항목의 모임을 포함하며 그뒤에는 두마디사이로 갈라 진 꼬리들의 실례를 표현하는 두개의 직접항목을 가지는 2개까지의 직접항목들의 모임이 놓인다. 만일 꼬리가 양식화된 머리부에 일치시킬수 있는 파일의 최대수보다 더 크면서 양식화되지 않은 마디크기(4K)보다 작으면 양식화되지 않은 마디에 기억되며 리용된 공간의 크기를 합한 지적자는 간접항목에 기억된다.

등록부들은 등록부항목들의 모임으로 구성되는데 등록부항목들은 등록부입구점들의 모임으로 이루어 진다. 한편 등록부입구점들은 파일이름과 이름으로 된 파일의 열쇠를 포함한다. 한개의 단일한 마디에 기억된 같은 객체에는 동일한 항목형을 가지는 하나이상의 항목은 존재하지 않는다(결합된 항목보다 두개로 분리된 항목을 리용하려고 할 리유는 없다.).

균형을 유지하려고 할 때나 마디에 그것의 린접한 마디를 묶는 문제를 해석해 보면 ReiserFS는 세개의 마디를 두개 마디로 줄일수 없다는것을 시사해 준다.

나무의 순서화

일부 열쇠정의방법은 패턴의 리용에 의거하고 있으며 이것은 파일체계를 생성할 때 여러가지 열쇠정의형식으로부터 한개를 아무때나 선택할수 있다는것을 의미한다. 실례로 모든 등록부입구점들을 파일체계의 앞에 묶어 놓겠는가 혹은 파일의 이름에 가까운 입구점들로 묶어 놓겠는가를 결정하는 방법을 고찰하자. 대규모파일에서는 리용패턴을 가지고 있는 체계가 등록부의 모든 입구점들을 변경시키는데서 효과적이므로 모든 등록부항목들을 함께 선택하여야 한다. 작은 파일에 관하여서는 이름이 파일에 가까와야 한다. 류사하게 대규모파일에서 정적자료는 같은 등록부로부터 서로 다른 정적자료라든가 혹은 등록부입구점들을 가지는것은 본체와 따로따로 기억되여야 한다.

이러한 알고리듬을 리용하는 의미론적문법객체를 완전히 독립적으로 묶어 놓을수 있으며 객체의 이름들에 의하여 결정되는것과는 다른 묶음방법이 더 적합한 응용프로그람도 있을수 있다.

열쇠의 구조

매개 파일항목은 구조체 즉 locality_id, object_id, offset로 된 유일성마당을 가진 열쇠이다. 기정값에 의하여 locality_id는 상위등록부의 object_id이다. object_id는 파일의 유일한 id이며 객체가 생성될 때 리용되지 않은 첫번째 object_id에 설정된다. 이것은 서로 이웃에 묶어 지고 있는 등록부안에 객체가 성공적으로 생성된 결과이며 패턴리용방법의 우점으로 된다. 파일에서 변위는 항목의 첫번째 바이트로 되는 론리적객체안에 있다. 판본 0.2에서 모든 등록부입구점들은 그 입구점에 기억된 자체의 개별적열쇠를 가지고 있으며 매개는 항목별로 서로 구별된다. 현행 판본에서는 ReiserFS가 첫번째 입구점의 열쇠로 되는 항목에 한개 열쇠를 기억하며 이 한개의 열쇠로부터 요구되는만큼 매개 입구점의 열쇠를 계산한다. 등록부에서 변위열쇠성분은 파일이름의 첫 4개 바이트이다. 따라서 수값

적인 변위라기보다 사전편찬적인 내용으로 이것을 생각할수도 있다. 등록부항목에서 유일성마당은 첫 4개 바이트안의 동일한 파일이름입구점들을 구별한다. 모든 항목들에서 유일성마당은 항목의 형을 지시하며 완충기안의 제일 왼쪽항목에 관하여 나무안에 앞서는 항목이 같은 형이나 객체중에 있는가를 지적한다. 열쇠에 이 정보를 주는것은 균형조건을 해석하는데서는 쓸모가 있으나 비등록부항목에 관해서는 열쇠의 길이가 늘어 나게 되며 방식적특성에 의문을 가지게 한다.

매개 파일은 유일한 object_id를 가지지만 객체를 찾는데서는 리용할수 없으며 여기에는 오직 열쇠만을 리용할수 있다. object_id는 순수 열쇠가 유일한가만을 확인한다. 만일 우리가 객체의 열쇠를 변경시키는 ReiserFS특성을 사용할수 없다면 불변특성을 가지며 그렇지 않다면 변경특성을 가지게 된다(이 특성은 NFS데몬 등에서도 지원된다.). 개발자들은 객체를 식별하는데 변경열쇠를 리용하는 문제가 구성방식적으로 해로운 결과를 초래하지 않겠는가 하는 문제를 가지고 오랜 기간 론의하였다. 한 연구자는 열쇠를 기록하는 임의의 객체가 자체복사를 갱신할수 있는 방법을 가져야 한다는 요구가 제기되면 해결할수 있다고 보았다.

이것은 object_id가 객체의 의미론적표현을 변화시키지 않으므로 참조위치가 매우 불명확해 지는 위치에 object_id의 넘기기를 캐쉬에 기억해야 하는 현상을 극복하기 위한 방식도로 된다. 이 연구자는 열쇠가 명백하게 변경되지 않는 한 첫번째로 생성된 등록부의 위치를 꾸려 묶는 방법으로 객체들을 묶음화하였다. 이 묶음은 등록부로부터 련결이 해제된다고 해도 묶어 진채로 그냥 남아 있게 된다. 이것은 모든 다중련결파일들을 기억하는 일부 다른 파일체계와는 달리 명백한 요청이 없이 생성되는 위치로부터 옮기지 못하고 두번째 련결이 이루어 질 때 최초의 위치로부터 옮겨야 했다. 다중등록부와 련결된 파일은 적어도 그 등록부들의 하나에 리로운 위치기준을 얻어야 한다. 요약하여 말하면 이 방법은 첫째로 같은 등록부로부터 파일들을 배치한 다음 같은 등록부로부터 등록부에 관한 통계자료로서 등록부입구점들을 배치한다. 순서화된 서로 다른 등록부로부터의 간격배치는 없으며 같은 등록부안의 모든 등록부입구점들은 련속적으로 배치된다는데 대하여 강조해 둔다. 이것은 실제적으로 공통선조를 가지는 작은 등록부안의 파일들을 꾸려 묶지 않으며 선형순서화를 결정하는데서 완전부분순서화를 리용하지 않는데 여기서는 순수 상위등록부정보를 리용한다. 완전한 나무구조에 대한 지식을 리용하는데서 적절한 경우는 FS cleaner의 실현에 있으며 동적알고리듬에는 없다.

나무마디의 균형화는 다음의 순서화된 우선권에 따라 이루어 진다.

1. 사용된 마디의 수를 최소화한다.
2. 균형조작에 의하여 영향을 받는 마디수를 최소화한다.
3. 균형조작에 의하여 영향을 받는 캐쉬에 기억되지 않는 마디수를 최소화한다.
4. 만일 다른 양식화된 마디를 옮기려면 우선권순위에 의하여 옮겨 진 바이트들을 최대화한다.

또한 보다 미묘한 효과들도 있다. 만일 서로 다른 마디다음에 마디를 임의로 놓고

무엇이든 효과적으로 묶어 진 마디들사이라든가 혹은 극단적으로 즉 잘 묶어 졌거나 혹은 걸써하게 묶어 진 마디들사이에서 어떤 극단적방략을 취하면 마디들을 더많이 결합할수 있다. 이때 극단적인 옮김조작은 내부마디에 대하여서는 적당치 않다.

완충화와 보존목록

ReiserFS의 0.2판본은 나무에서 모든 항목의 이동을 추적하는 쓰기순서화체계를 실현하였다. 이 판본은 항목을 접수한 마디가 씌여 지기전에 항목이 옮겨 진 마디가 씌여 질수 없다는것을 확인하였다. 이것은 최근에 생성되지 않은 항목의 루실을 발생하는 체계폭주를 막을수 있으므로 필요하다. 이 추적방법은 이미 연구되였으며 그것을 강요한 휴지시간은 현재의 성능평가기준에 의하여 측정할수 없었다. 이다음 판본에서 항목의 이동을 부분적으로 변화시키고 항목의 형개수를 증가시켰을 때 이 코드는 그것의 복잡성으로 하여 조종에서 벗어 났다. 이 코드를 보다 더 간단한 도식으로 바꾸기 위한 연구가 심화된 결과 전형적인 리용패턴에서는 더 좋은 효과가 얻어 졌다.

이 도식은 다음과 같다. 항목이 마디로부터 옮겨 지면 그것의 완충기가 씌여 질 블로크를 변경한다. 그 블로크를 왼쪽 근방의 변경된 블로크들에 대하여 가장 가까운 자유블로크로 변경시키며 그렇지 않으면 그 블로크를 비우고 《보존목록》에 변경된 블로크번호를 기입한다(여기서 가장 가깝다는 말은 아주 단순하다. 블로크번호지정함수가 블로크번호가 증가하는 영향의 왼쪽근방으로 옮긴다는 의미이다.). 《일치순간》이 얻어 질때 보존목록의 모든 블로크들을 비운다. 일치순간은 객체가 옮겨 진 기억에 마디가 하나도 없을 때 발생한다. 만일 디스크공간을 벗어 나면 일치순간발생을 요구한다. 이 조작은 파일체계가 회복가능하다는것을 확인할 때 효과적이다. 대규모파일의 성능평가를 진행하는 동안 성능평가도중에 보존목록이 몇번 비워 졌다는데 대하여 강조해 둔다. 보존된 완충기의 퍼센트값은 지우기동안을 제외하고는 실천적으로 작으며 필요한만큼 자주 일치순간이 발생할수 있게 미리 준비할수 있다.

이 방법은 BSD의 쏘프트갱신방법(Soft Updates approach of BSD)이나 ReiseFS 0.2판본에 의한 방법보다는 좀 못할수도 있다. 그러나 그러한 쓰기순서추적방법은 항목을 부분적으로 옮기는 균형나무에 관한 방법보다 더 복잡하다. 그러나 ReiserFS는 1~10K크기의 범위에 있는 파일에 대하여 실제상 성능에 장애가 있을수 있기때문에 앞으로는 변경된 알고리듬으로 될것이다.

ReiserFS구조

ReiserFS나무는 Max_Height=N(현재 기정값은 N=5)의 최대높이를 가지며 디스크블로크에 존재한다. ReiserFS나무에 속하는 매개 디스크블로크는 한개 블로크의 머리부를 가진다.

파일체계안의 매 객체는 항목의 모임으로 기억되며 매 항목은 item_head를 가진다. item_head는 항목의 열쇠와 그것의 빈 공간(간접항목에 대한)을 포함하며 블로크안의 항목자체의 위치를 정의한다. 나무의 내부마디를 포함하는 디스크블로크는 열쇠와 디스크블로크에 대한 지적자이며 그 형식은 다음과 같다.

Block_Head Key Key Key … Key Pointer Pointer … Pointer … Pointer … 빈공간
0 1 2 N 0 1 N N+1 …

나무안의 매개 잎(잎은 *n*개의 항목과 대응하는 머리부를 가진다.)은 다음과 같은 내용을 포함하는 디스크블로크를 가진다.

Block_Head 1Head 1Head … 1Head Point … 빈공간 Item … Item Item
0 1 N 0 … N 1 0

또한 우에서 언급한 나무의 양식화되지 않은 마디를 포함하는 디스크블로크도 있다. 이 종류의 디스크블로크들은 자료를 포함하는데 밖으로부터는 구조적으로 비여 있는것처럼 보인다(자료를 포함하고 있다고 해도).

* * * * * * * * * *

ReiserFS의 이름공간(파일과 등록부를 포함하는)에서 객체의 최대수는 다음의 방법으로 계산한다.

$$2^{32}-4$$

이 값은 4,294,967,292와 같다.

내부마디구조 다음의 표에서는 디스크상에 기억될 때의 색인마디블로크구조를 보여준다. ReiserFS 색인마디는 곧 열쇠와 디스크자료블로크들에 대한 지적자를 기억하는 ReiserFS나무의 한개 마디이다.

Block_Head Key Key Key … Key Point Point Point … Point Point … 빈공간
0 1 2 N 0 1 2 N N+1 …

여기서 우리는 열쇠구조에 관한 표현을 볼수 있다. 모든 변수들은 32bit크기를 가진다는것을 강조한다.

마당이름	형	크기(byte)	설 명
K_dir_id	_u32	4	상위등록부의 ID
K_object_id	_u32	4	객체의 ID(색인마디의 번호)
K_offset	_u32	4	객체의 시작으로부터 현행바이트까지의 변위
K_uniqueness	_u32	4	항목의 형(통계자료=0, 직접=−1, 간접=−2, 등록부=500)
	총	16	내부마디에 대하여 (6) 8byte 잎마디에 대하여 (22) 24byte

마지막으로 디스크블로크에 대한 실제적지적자인 disk_child구조체를 보기로 한다.

파일이름	형	형크기(byte)	설 명
dc_block_number dc_size	unsigned long unsigned short	4 2	하위디스크의 블로크번호 하위디스크에 사용된 공간
	총	6	(6) 8바이트

잎마디구조 아래의 표에서 보여 준바와 같이 ReiserFS는 나무의 한개 마디이며 항목과 그것의 머리부를 기억하는 디스크블로크에 대하여 해석해 보겠다.

항목에는 통계자료항목, 등록부항목, 간접항목, 직접항목이 있는데 이 경우에는 자체해석에 속한다.

Block_Head	IHead 0	IHead 1	…	IHead N	Poiuter 0	빈공간	Item N	…	Item 1	Item 0

이제 디스크블로크의 block_head로부터 시작하여 개별적객체를 걸쳐 다시 계속한다.

마당이름	형	크기(byte)	설 명
blk_level	unsigned short	2	나무의 블로크준위(1－잎；2, 3, 4 …내부)
blk_m_item	unsigned short	2	내부블로크의 열쇠번호 혹은 잎블로크의 항목수
blk_free_space	unsigned short	2	나이트로 된 블로크빈공간
blk_right_delim_key	struct key	16	해당 블로크의 오른쪽 뿌리열쇠
	총	22	(22) 24 잎마디에 대한 바이트수

매개 항목머리부는 항목의 항목나무안에서 정착위치를 찾고 사용된 공간뿐아니라 항목에 남아 있는 빈 공간에 대한 몇가지 정보도 얻을수 있게 하는 여러개의 변수를 포함한다.

마당이름	형	크기(byte)	설　명
ih_key	struct key	16	항목을 찾기 위한 열쇠, 모든 항목머리부는 이 열쇠에 기억된다.
u.ih_free_space	_u16	2	간접항목을 위한 마지막양식화되지 않은 마디의 빈 공간, 직접항목에 대해서는 OxFFFF, 통계자료에 대해서는 OxFFFF, 등록부항목의 등록부입구점수
ih_item_len	_u16	2	항목본체의 총 크기
ih_item_location	_u16	2	블로크안의 항목본체에 대한 변위
ih_reseved	_u16	2	ReiserFSck에 의하여 리용된다.
	총	24	24byte
sd_mode	_u16	2	파일형, 허용
sd_nlink	_u16	2	고정련결수
sd_uid	_u16	2	소유자 id
sd_gid	_u16	2	그룹 id
sd_size	_u32	4	파일크기
sd_atime	_u32	4	마지막접근시간
sd_mtime	_u32	4	파일의 마지막변경시간
sd_ctime	_u32	4	색인마디(통계자료)의 마지막변경시간 (sd_atime과 sd_mtime의 변화를 제외하고)
sd_rdev	_u32	4	장치
sd_first_direct_byte	_u32	4	파일의 시작으로부터 파일의 직접항목의 첫 바이트까지의 변위(－1) 작은 파일용(파일이 직접항목만 가진다.)의 등록부에 대하여 (>1), 큰 파일용(파일이 간접항목과 직접항목을 가진다.)에 대하여(－1), 큰 파일(파일이 간접항목은 가지지만 직접항목을 가지지 않는다.)
	총	32	32byte

작은 파일이든지 큰 파일이든지 등록부객체는 반드시 파일이름을 포함한다.

deHead	deHead	deHead	－deHead	filename	－filename	filename	filename
0	1	2	N	N	2	1	0

작은 파일들은 한개의 지적자에 의하여 주소화되기때문에 직접항목(direct item)이라고 한다.

＊ ＊ ＊ 작은 파일본체 ＊ ＊ ＊

보다 큰 파일(한개이상의 블로크디스크가 필요한)들에는 모든 련속된 블로크들을 찾기 위하여 간접항목(indirect item)이라고 부르는 기교적인 여러개 지적자가 필요하다.

unfpointer0 unfpointer1 unfpointer2 … unfpointerN

좀 더 구체적으로 설명하면 unfpointer는 큰 파일의 본체를 포함하는 양식화되지 않은 블로크에 대한 지적자(32bit)를 포함한다.

다음 표에서 지적자가 양식화되지 않은 블로크를 어떻게 찾는가를 알수 있다.

마당이름	형	크기(바이트)	설 명
deh_offset	_u32	4	등록부입구점열쇠의 세번째 성분(이 값에 의하여 모든 ReiserFS_de_head가 정렬된다.)
deh_dir_id	_u32	4	객체의 상위등록부의 object_id등록부입구점에 의하여 참조된다.
deh_object	_u32	4	등록부입구점에 의하여 참조되는 객체의 object_id
deh_location	_u16	2	전체 항목안의 이름의 변위
deh_state	_u16	2	1) 입구점은 통계자료를 포함한다. 2) 입구점은 완페된다.
	총	16	16byte

여기서 파일이름은 파일의 이름을 의미한다(리용가능한 길이의 바이트배렬). 파일이름의 최대길이=블로크크기－64(4KB의 블로크크기에 관하여 최대이름길이는 4032byte이다.)

파일배치최량화에 나무의 리용

4개 준위 파일배치(Layout)최량화가 수행된다.

▼ 론리적블로크디스크상의 물리적위치에로의 넘기기
- 론리적블로크번호에 대한 마디의 지적
- 나무에서 객체들의 순서화

▲ 객체가 묶어 지는 마디를 통한 객체의 균형화

물리적배치(Physical Layout)

디스크상에서 론리적블로크번호의 물리적위치에로의 넘기기는 SCSI용디스크구동기 제조자들과 IDE용장치구동기에 의하여 실현된다. 물론 앞의 장들에서 고찰한 LVM과 같은 보다 높은 준위의 프로그람도 있을수 있는데 이런 프로그람들은 넘기기를 더 높은 수준에서 추상화한다. 구동기제작자들에 의한 론리적블로크번호의 물리적위치에로의 넘기기는 보통 실린더(cylinder)에 의하여 진행된다. ReiserFS개발자들은 실린더경계를 추적하지 않고도 의미론적으로 린접한 론리마디들의 거리최소화를 실제 실린더경계에 따라 근사적으로 실현할수 있었다. 이리하여 보다 좋은 체계를 실현할수 있게 되였다.

마디배치(Node Layout)

ReiserFS는 디스크에 나무의 마디들을 배치할 때 블로크번호에 리용된 비트매프에서 첫번째 자유블로크를 찾는다. 이때 나무의 순서화에서 왼쪽린접마디의 위치로부터 시작하여 마지막으로 옮겨 진 방향으로 이동시킨다.

실험에 의하면 이 방법이 다음의 성능평가(benchmark)선택법보다 더 효과적이라는것을 보여 주었다.

1. 비트매프에서 첫번째 령 아닌 입구점을 취한다.
2. 마지막으로 옮겨 진 방향을 지적하는 입구점뒤의 입구점을 취한다(이 방법은 쓰기속도는 3%정도 높지만 련속읽기에서는 10~20%정도 낮아 진다.).
3. 왼쪽린접에서 출발하여 오른쪽린접방향으로 옮긴다. 옮겨 진 항목들이 디스크를 새로운 접수마디(보존목록을 보기 바란다.)에 도달시키기전에 전송마디의 반복쓰기를 피할 목적으로 블로크번호를 변경시킬 때 여기에 리용된 성능평가기준은 설사 왼쪽린접을 결정하는데서 적지 않은 휴지시간이 추가된다 해도 마디의 현행블로크번호보다도 오히려 왼쪽근방으로부터 찾기를 시작할 때보다 대략 10%정도 더 빨랐다(현재 실현수법은 왼쪽근방의 상위객체를 읽을 때 I/O에 위험을 조성한다.).

ReiserFS는 디스크구동기의 끝에 도달할 때 방향을 바꿔 주는데도 리용된다. 개발자들은 파일에 블로크들을 배정할 때 그것이 움직이는 위치에 차이가 있는지 없는지를 알기 위하여 검사를 진행하였으며 블로크번호가 증가하는 방향으로 배정할 때 항상 현저한 차이가 생긴다는것을 알게 되였다. 이것은 블로크번호의 증가에 대한 배정조작에 의하여 디스크회전위치를 정합시킨데 기인될수도 있다.

선행가동일지기록(write-ahead logging)

대다수의 메타자료조작들은 한개이상의 블로크를 포괄하며 메타자료는 보통 한개 조작에 포함된 일부 블로크들만이 갱신되면 손상되게 된다. 선형가동일지기록을 리용하면 블로크들은 실지위치에 일치되기전에 가동일지에 기록된다. 폭주후에 가동일지는 파일체

계를 정상상태로 회복시키기 위하여 재생(replay)된다. 이 재생조작은 fsck로 하는것보다 훨씬 빠르며 그 시간은 파일체계의 크기보다 가동일지구역의 크기에 의하여 한정된다.

ReiserFS실행기록특성

실행기록은 몇가지 종류의 가동일지조작과 그것의 잠금을 위한 직렬화를 요구한다. 이 리유는 실행기록된 파일체계가 필수적으로 가동일지에 기록되지 않은 복제본보다 속도가 뜨다는데 있다. 따라서 실행기록은 그러한 가동일지등록부동작과 직렬화를 수행하기 위한 몇가지 종류의 기본적인 조작을 요구하게 된다.

ReisserFS에 어떤 조작들이 있는가를 보자.

거래(transactions)

실행기록방식에서 매개 조작은 서술블로크를 포함하며 그뒤에 많은 자료블로크와 결속블로크(commit block)를 포함한다. 서술블로크와 결속블로크는 매개 론리적블로크에 해당한 실지디스크위치로 구성되는 순서목록을 포함한다. 가동일지는 갱신되는 순서로 보존되여야 하며 만일 쓰기프로그람이 블로크 A, B, C, D를 가동일지에 기록하고 다시 A를 기록하면 B, C, D, A순서로 가동일지에 순서화된다. 블로크가 결속되지 않은 트랜잭션에 있을 때 그 블로크는 지워 져 있어야 하며 적어도 하나의 참조계수값을 가지고 있어야 한다. 일단 한개의 거래가 디스크상에 모든 가동일지블로크들을 가지고 있기만 해도 실제완충기들은 낡아 지게 되여 개방된다.

묶음식거래

다중거래들은 단일한 원자적단위로 결합할수 있다. 만일 거래 한개가 블로크 A, B, C를 가동일지에 기록하고 거래 두개가 블로크 A, C, D를 기록하면 련결된 결과 거래는 서술블로크를 쓰고 다음 블로크 B, A, C, D를 쓰며 그 다음 결속블로크를 쓴다. 이 조작은 좀 더 많은 전체 블로크들이 가동일지에 기록되게 할수 있지만 파일체계가 폭주되여 동작하지 않고 변화되는 경우가 많아 질수 있다. 거래가 언제 어떻게 서로 묶음화되는가를 조종하기 위한 많은 이행파라메터들이 있다. 이제 그 이행부분에 대하여 구체적으로 보자.

비동기결속 비동기결속은 묶음식거래의 확장이다. 거래는 디스크에 대하여 모든 가동일지블로크들을 소거함이 없이 완료되게 할수 있다. 이 조작은 좀 복잡하지만 보다 빨리 귀환되게 하며 블로크들이 유지하고 있는 임의의 잠금을 개방할수 있게 한다. bdflush는 디스크에 대하여 변경된 비동기적가동일지블로크를 자연스럽게 해결한다.

새 블로크는 캐쉬에서 제거할수 있다. 만일 블로크가 배정되고 디스크에 기록되기전이나 가동일지에 기록되기전에 비워 지며 다음 그 거래가 완성되기전에 비워 지면 블로크는 절대로 가동일지에 등록될수 없거나 혹은 실제디스크에 기록될수 없다. 이 현상은 많은 파일체계들에 고유하지만 ReiserFS에서 옳게 리해하자면 좀 더 연구해야 한다.

선택적인 소거기능 이 조작은 실제적으로 특성보다 요구가 더 많다. 많은 블로크들(비트매프, 상위블로크)은 몇번이나 다시 가동일지들에 등록될수 있다. 블로크가 결속되

지 않은 거래에 있을 때는 낡아 질수 있으며 디스크에 전송될수 있다. 그러나 가동일지 구역이 재사용가능하면 거기에 포함된 임의의 거래는 실제위치에 소거된 모든 블로크들을 가지고 있어야 한다.

다중으로 가동일지에 등록된 블로크가 어떤 방법으로든지 소거될수 있다고 할지라도 그 블로크는 다른 모든 블로크와의 관계에서 더 변경된 거래로 변화되며 폭주후에 메타자료가 손상될수 있다. 차후 거래로 될수 있는 블로크들을 소거할 대신에 거래의 가동일지를 디스크에 강하게 요구한다.

폭주후에 모든 요소들이 일관화될수 있게 가동일지가 재생된다. 이것은 자주 가동일지에 기록되는 블로크들은 오직 한번만 매 거래당 가동일지에 등록되여야 하며 파일체계의 올려태우기해제에 관하여 블로크들의 실지위치에 한번만 기록되여야 한다는것을 의미한다.

자료블로크가동일지기록(Data Block logging) 지금 블로크들은 직접항목부분이 간접항목에 일치시킬수 있다는 사실을 초월하여 직접항목을 포함하는 파일의 mmap우에서 확장될 때 진행된다. 때로는 변환이 변경된 자료에 대하여 진행되기때문에 폭주후에 자료가 잃어 지지 않았는가를 확인하여야 한다.

문제는 블로크가 일단 가동일지에 있기만 하면 폭주후에 블로크들을 중복쓰기할 가동일지의 재생기회가 있는 동안은 가동일지등록을 계속해야 한다는데 있다. 그리하여 mark_buffer_dirty가 호출되는 일은 없게 되며 대신 저널의 호출은 블로크가 현재 거래에 있거나 혹은 재생될수 있는 거래일 때만 블로크의 가동일지기록목적으로 존재하게 된다. 이것은 가능한 꼬리들을 리용하여 평균적인 크기로 파일들을 일치시킨다는것을 말한다. 왜냐하면 많은 자료블로크들이 가동일지에 기록될것이기때문이다.

파일닫기상태에서 묶음파일을 실현하는 방법과 가능한 꼬리들을 리용하지 않고 올려태우기를 실현하는 해결안도 있다. ReiserFS개발그룹은 대체로 이런 문제점에 대하여 고찰했을것이다.

전환(Tuning) 가동일지구역의 크기는 성능에서 제일 큰 영향을 준다. 이 구역을 너무 작게 하면 너무 자주 그것의 실제위치에 대한 블로크의 flush가 진행되여야 한다. 너무 크게 하면 재생시간이 지내 오래 걸리게 된다.

그렇다면 정확한 크기를 어떻게 찾아 낼수 있는가? beta1에서 성능평가기준을 될수록 빨리 얻을수 있는 정도까지 시험해 보아야 한다. Bate2는 실지블로크들을 flush할 때 올려태우기선택에 정보적인 명령문들을 추가할수 있다. 이 방법들은 보다 더 쉽게 전환될수 있어야 한다. 변경된 객체들을 어떻게 얻을수 있겠는가와 관련된 최대거래크기, 최대묶음크기(betch size), 여러가지 시간제한들도 역시 중요하다.

드롭(ReiserFS Drops) 매개 드롭이 개별적인 열쇠를 가지는 드롭들로 한개 파일이나 등록부를 분할하는 문제를 고찰한다. 여기서 한개 드롭으로 압축하지 않고 같은 마디를 차지하는 파일이나 등록부로부터 분할되는 두개의 드롭은 존재하지 않는다. 매개 드롭의 열쇠객체에 대한 열쇠와 거기에 객체안의 드롭의 변위를 더한것이다. 등록부에서 이 변위는 사전적의미를 가지며 수자적이며 바이트로 된 파일에 관하여 파일이름에 의거한다. 몇가지 파일체계판본과정에 액체, 고체, 공기방울을 가지고 실험을 진행하였으며 실현하였다. 고체덩이는 옮겨 질수 없고 그 방울은 양식화된 마디의 전체를 차지할 때 굳어 지기만 한다. 액체방울은 마디를 완전하게 늘구는 임의의 액

체방울이 마디의 공간을 차지하는 방법으로 옮겨 지게 된다. 물리적액체와 마찬가지로 액체방울은 옮길수는 있지만 압축할수 없다. 공기방울은 순전히 나무의 균형화조건에 맞는다.

ReiserFS 02는 파일전체에 대하여서는 액체방울을 실현했지만 파일의 꼬리에 대하여서는 그렇게 하지 않았다. 파일이 적어도 크기상으로 한개 마디이면 마디의 시작점으로 파일의 시작점을 파일의 블로크정돈으로 맞출수 있을것이다. 다중드롭파일의 시작점에 대한 이러한 블로크정돈은 공간을 랑비하는 설계오유였다. 만일 기준위치가 의미론적으로 린접한 파일부분을 읽을수 없을 정도로 불투명하면 그리고 마디들이 서로 가까우면 여유블로크를 읽는 비용은 파일의 첫 마디에 도달하기 위한 찾기와 회전비용에 의하여 철저하게 작아 지게 될것이다. 비용은 4~20K의 파일들에 해당한 일정한 공간으로 되지만 결과적으로 블로크정돈에는 아주 적은 시간이 소비된다.

다중드롭파일의 블로크정돈을 포함하는 ReiserFS와 비간접항목들은 13K의 크기로 평균화된 파일에 관하여 88%공간효률만을 초래한 흥미 있는 동작을 실험적으로 얻게 되였다. 4K보다 더 큰 파일꼬리가 순서화된 나무에서 4K보다 더 큰 다른 파일보다 앞에 놓이면 드롭이 먼저 굳어 지고 정돈되며 후에 굳어 지고 정돈되기때문에 꼬리가 있는 크기라고 할지라도 그것은 전체 마디에 배치된다.

현재 일부 판본에서는 큰 파일들의 꼬리부분을 양식화되지 않는 옹근마디들에 예약된 나무준위로 배치하고 양식화되지 않는 마디들을 지적하는 양식화된 마디의 간접항목들을 생성한다. 이 내용은 자료기지항목에서 하나의 연구방법으로 알려 져 있다. 나무에 첨가된 이 여유준위는 더 적게 균형화된 나무(지적된 양식화되지 않는 마디들을 나무의 부분으로 고찰한다.)를 생성하는 어용으로 되며 그것에 의하여 나무의 최대깊이를 증가시키는 어용으로 된다.

중간크기의 파일에 대하여 간접항목을 리용하면 그 항목들을 가진 자료의 혼합에 의하여 지적자의 캐쉬비용을 증가시키게 된다. 전개의 축소는 흔히 읽기알고리듬이 어떤 시각에 한마디만을 불러 내는 결과를 초래한다. 왜냐하면 파일자료를 가지고 있는 마디를 읽기전에 캐쉬에 기억되지 않은 간접항목을 읽으려고 대기하기때문이다.

마디에서 꼬리와 간접항목의 혼합에 의하여 전개가 줄어 드는 직접적인 결과로서 내부마디의 리용보다 간접항목의 리용에 의거하여 읽기를 진행하는 매 파일당 여러개의 선조가 있다. 가장 치명적인 결함은 파일의 읽기에 필요한 여러개 마디들의 이러한 읽기가 드롭들에 비교될 정도의 추가적인 찾기와 회전을 요구한다는것이다.

초기드롭을 연구한데 의하면 드롭들은 보통 디스크의 배치설계나 심지어 꼬리까지도 순차적이며 내부마디의 선조들은 그 선조나 캐쉬에 있는 다른 내부마디에 포함되여 있는 모든것들이 한번에 하나의 순차적읽기로 요청될수 있는 방법으로 그것들을 지적한다.

마디의 비순서화된 읽기는 순서화된 읽기보다 비용이 더 들며 이와 같은 단순한 고찰은 효과적인 읽기최량화를 요구한다. 양식화되지 않은 마디들은 파일체계의 회복을 더 빨리 그리고 적은 로바스트성으로 하게 하며 이때 파일체계는 복구된 나무에 그것을 삽입할 대신 그것들의 간접항목을 읽으며 한편 그 내용이 파일로부터 생성된다는것을 확신할수 없다. 이 방법들은 ReiserFS를 가동일지등록이 없는 색인마디기초파일체계와 류사하

게 만들어 준다. 현대적이며 보다 개선된 해결방안은 BLOB를 도입하는것보다도 마디의 시작점에 다중마디파일의 시작점을 배치하기 위한 요구를 취소하는것이다. 또한 이 해결방안은 고체덩이들을 이동시키는 알고리듬을 리용하여 빈번히 이동하지 않는 파일의 80%를 묶어 주기 위한 파일체계소거프로그람을 리용하는것이다. 이 방법은 여전히 mmap()의 목적에 효과가 적은 양식화된 마디들에 대하여서는 문제로 남아 있다(이 방법은 페지표입구점들을 단순하게 변경시키는것보다는 오히려 쓰기전에 그것을 복사해야 하며 기억대역근 CPU의 어용이 나가자고 해도 어용상 비싸 진다.).

이 리유에 대하여 다음 판본을 위한 계획을 가지고 설명한다. 이제 3개의 나무를 도입하겠다. 한 나무는 열쇠를 양식화되지 않은 마디들에 넘기며 다른 나무는 열쇠를 양식화된 나무에 넘긴다. 또 다른 나무는 열쇠를 등록부입구점들과 통계자료로 넘긴다.

이 방법은 파일, 등록부입구점의 첫번째 접근, 통계자료, 양식화되지 않은 마디 그리고 꼬리부분을 읽기 위하여 처음에 한 나무를 찾아 가고 다음에 다른 나무를 찾아 가는 방법으로 디스크량단사이거리를 길게 뛰여 넘어야 하는것처럼 생각할수 있다. 이때 계획은 다음의 알고리듬에 따라 세개 나무의 마디들을 끼워 넣는것이다. 블로크번호들은 마디가 생성되거나 혹은 보존될 때 마디에 대하여 지적되며 어떤 때는 소거프로그람이 실행될 때 지적될수 있다. 블로크번호의 선택은 첫째로 가깝게 배치되여야 할 다른 마디가 어느것인가에 기초하며 다음으로 승강기의 현행방향에서 제일 가까운 자유블로크를 찾는데 기초한다. 현재 우리는 가까이 배치되여야 할 마디로서 나무의 왼쪽근방마디를 리용한다. 새로운 도식은 먼저 가까이 배치되여 있는 마디를 결정하며 다음으로 그 점에서부터 자유블로크에 대한 탐색을 시작하는데 어느 마디를 가깝게 배치하는가에는 보다 복잡한 결정방법을 리용한다.

이 새로운 방법은 동일한 묶음위치로부터 모든 마디들이 서로 가까와 지게 하며 모든 등록부입구점들과 통계자료들이 그 묶음위치에 서로 그룹화되도록 하며 같은 묶음적위치로부터 양식화된 마디와 양식화되지 않은 마디를 끼워 넣게 한다. 가상코드가 이것을 서술하는데서는 제일 좋다.

코드복잡성 이제 코드길이설정에서 의의가 있는 몇가지 설계방법들에 대하여 언급하겠다. 보다 단단한 마디들을 묶을수 있게 항목의 일부만을 옮기는 균형알고리듬을 작성하면 코드의 복잡성을 피할수 없다. 다른 중복적이고 결정적인 코드균형화의 복잡성은 항목형들의 개수였다. 2중화된 간접항목의 도입과 액체방울으로부터 공기방울에로의 등록부항목의 변화도 역시 복잡성을 증대시켰다. 파일의 직접 혹은 간접항목에 통계자료를 기억시키는것은 통계자료를 그것의 자체 항목형으로 만들었을 때보다도 항목들을 처리하기 위한 코드는 복잡해 졌다. N×N코드화복잡성문제를 가지고 론할 때 이것은 보통 추상화계층을 한개 더 보충할것을 요구한다. 균형화된 코드복잡성에 관하여 항목수의 N×N효과는 그 설계원리에 대한 하나의 실례이며 우리는 그것을 다음의 기본재쓰기에서 주소화하게 된다.

균형코드는 모든 항목의 형을 다 지원하는 항목조작모임을 리용한다. 다음 균형코드는 항목정의형연산조종자(item_specific item operation handler)를 결정하기보다 항목의 형에 대한 그 어떤 여러개의 의미를 리해할 필요가 없이 이 조작들을 호출하게 된다. 새로운 항목의 형(례를 들어 압축항목)을 추가하는것은 현재 수행되는 균형코드의 많은 부분을

변경시킬것을 요구하기보다는 오히려 단순하게 그 항목에 관한 항목조작모임을 서술하게 할것이다.

이제 균형조작을 수행하기 위하여 어떤 원천들이 요구되는가를 결정하는 함수 fix_nodes()를 어떤 방법으로 쓰든지 편리하므로 균형조작시 어떤 조작이 수행되는가를 결정하는데 이 함수를 리용할것이다. 한가지 방법 즉 잠금이 된 마디를 가지고 균형동작을 수행하는 함수 do_balance()가 복잡성을 없앨수 있다.

Linux핵심부에서 ReiserFS의 설치와 배치구성

ReiserFS는 설치하기 아주 쉽다. Linux Kernel 2.4.3으로부터 토발즈는 표준 Linux원천에 ReiserFS를 포함시켰다. 이것은 새로운 핵심부에 관하여서는 핵심부원천에 대하여 아무것도 할 필요가 없다는것을 의미한다. 이 핵심부는 ReiserFS에 의하여 콤파일할수 있게 준비되여 있다. 변경된 핵심부에 대하여서는 www. name sys.com Web싸이트로부터 검사수정을 얻기 위하여 리용되는 절차가 있으며 그다음 표준 Linux원천코드에 검사수정을 적용해야 한다.

Linux-2.2.x핵심부

Linux-2.2.x핵심부에 관하여 다음과 같은 단계를 거쳐야 한다.

1. 거울들중의 하나로부터 제일 마지막ReiserFS검사수정을 얻는다. linux-2.2.19-ReiserFS-3.5.32-patch.bz2을 얻으려고 하면 이것을 아무곳에나 넣는다. 실례로 /usr/src/2.2.19/
2. http : //www. kernel.org로부터 2.2.19의 핵심부원천을 얻은 다음 그것을 아무곳에나 넣는다. 실례로 /usr/src/2.2.19/linux. 이제 /usr/src/2.2.19/에서 “ls”지령을 수행하면 다음과 같은 결과가 나온다.
   # ls
   linuxlinux-2.2.19-ReiserFS-3.5.32-patch.bz2
3. ReiserFS를 핵심부에 적용한다.
   # cd/usr/src/2.2.19
   # bzcat linux-2.2.19-ReiserFS-3.5.32-patch.bz2 /patch-p0
4. Linux핵심부를 콤파일하고 ReiserFS지원을 설정한다.
   # cd/usr/src/2.2.19/linux
   # make mrproper ; make menuconfig
5. ReiserFS지원을 여기에 설정한다. 실험적특성들을 투입하는것이 필요하나 정확한 핵심부에 의존하여 다음의것을 리용한다.
   # make dep ; make bzImage
6. 배치구성시 ReiserFS support의 질문에 따라 y 혹은 *n*으로 대답한다. 배치구성 Web페지를 읽는다. ReiserFS를 모듈로서 설정하려면 다음의것을 실행한다.

make modules
make modules_install
앞으로 ReiserFS를 갱신할 때 모듈을 재콤파일해야 한다고는 생각하지 말아야 한다.
리론은 좋지만 주요하게는 파일체계에 대한 대면부가 전체 시간동안에 빨리 변경되기때문에 현실성은 적다. 모듈만을 재콤파일하여 발생되는 오유는 완전히 숨겨 질수 있으며 개발자들은 전체를 재콤파일하지 않기때문에 그것이 어디 있는가를 알수 있다. 핵심부의 이메지는 /usr/src/ 2.2.19/linux/arch/i386/boot/bzImage이다.

7. ReiserFS편의프로그람들을 콤파일하고 설치한다.
   # cd/usr/src/2.2.19/linux/fs/ReiserFS/utils
   # make ; make install
8. ReiserFS로 새로운 Linux핵심부이메지를 적당한 위치에 복사한다(보통 "/boot"등록부에 둔다.).
9. /etc/lilo. conf파일을 변경시키며 새로운 핵심부로 기동할수 있게 한다. lilo지령을 수행하고 lilo−21.6 혹은 더 새로운 파일을 리용한다.
   Lilo−21.6−or−newer
10. 구축된 핵심부를 기동하며 mkreserfs로 구획을 준비하며 그것을 올려태우기한다.
   # mkreiserfs/dev/xxxx
   # mount/dev/xxxx/mount−point−dir
   혹은
   # mount−t reiserfs/dev/xxxx/mount−point−dir
11. 작업을 진행한다.

Linux−2.4.0～2.4.2

ReiserFS코드는 Linux2.4.1−pre4의 Linux핵심부안에 있다.

1. Linux2.4.2를 얻는다. http : //www.kernel.org
2. 제일 마지막 ReiserFS−3.6.x patch를 얻는다.
3. 검사수정을 적용한다.
   # zcat linux−2.4.2−reiserfs−20010327−full.patch.gz / patch-p0
4. 핵심부를 콤파일한다(먼저 설명한것처럼 실험적특성들을 투입한다.).
5. 임의의 등록부에 tar압축을 해제한다.
6. ReiserFS편의프로그람을 콤파일하고 설치한다.
   # tar−xzvf reiserfsprogs−3.x.0i−1.tar.gz
   # cd reiserfsprogs-3.x.0i-1
   # ./configure
   # make ; make install
7. 구축된 핵심부를 기동하고 mkreiserfs 구획을 준비한다.

배치구성(configuration) ReiserFS기능성에 영향을 주는 콤파일시간선택항목이 있다. Linux핵심부의 배치구성단계를 수행할 때 이 선택항목들을 설정해 줄수 있다.

make config
make menuconfig
make xconfig

ReiserFS선택(option)항목을 핵심부배치구성으로 전환한다. 이 조작은 ReiserFS의 부모듈들을 구축한다.

ReisreFS를 구축한다. 이 조작은 ReiserFS를 핵심부방식이나 혹은 자립형핵심부방식으로 구축한다.

선 택 항 목	설 명
CONFIG REISERFS CHECK	핵심부배치구성시 이 항목을 yes로 설정하면 reiserFS는 이 조작을 통하여 내부일관성의 매개 가능한 검사를 진행한다. 가능한 검사를 진행하되 순차적으로 천천히 진행한다. 이 선택의 리용은 개발그룹으로 하여금 말단사용자에 대하여 그 영향을 근심하지 말고 오유수정시의 일관성을 검사할수 있게 한다. 만일 오유통보가 보내기직전에 있다면 yes로 대답하며 쓸모 있는 오유통보문을 얻을수 있다. 대다수의 사용자들의 경우에는 no로 대답한다.
CONFIG REIGERFS RAW	이 항목을 yes로 설정하면 등록부들을 우회하여 ReiserFS나무에 대한 행대면부를 제공하는 ioct1들의 모임을 가능하게 하며 변경된 파일들을 자동적으로 제거한다. 이 조작은 스쿠드캐쉬등록부를 위하여 설계된 실험적특성이다. Documentation/ filesystems/reiserfs_raw.txt를 보기 바란다. 이 기능은 ReiserFS를 back_end squid로 리용하기 위하여 특별히 설계되였다. 일반적착상은 모든 파일체계휴지시간을 피하며 ReiserFS 내부나무를 직접 찾을수 있다는것이다. 일반적핵심부에는 없다.
USE INODE GENERATION COUNTER	색인마디세대계층을 알아 내기 위한 3.6상위블로크의 s_inode_generation 마당을 리용한다. 정의되지 않으면 이 목적을 위하여 대역사건계수기를 리용한다(ext2과 대다수의 다른 파일체계에서 하는것처럼). 색인마디세대계층의 동작은 NFS에서 중요하다. 이 변수는 핵심부배치구성수속들을 통하여서는 리용할수 없다. include/linux/reiserfs_fs. h를 수동으로 편집한다.
REISERFS HANDLE BADBLOCKS	비트매프를 조종하기 위하여 ioct1을 허용한다. 이것은 잎랑블로크조종을 위한 원형으로 리용될수 있으나 실제적인 해결방법은 지금 진행중에 있다. 이 변수는 핵심부의 배치구성수속을 통하여서는 리용할수 없다. edit include/linux/reiserfs_fs.h를 수동으로 편집한다. 다음 리용가능한 올려태우기선택항목을 찾아 본다.

제１０장. 확장파일체계

Linux용의 가장 전망성 있는 새로운 실행기록파일체계의 하나는 SGI가 후원하는 XFS파일체계이다. XFS파일체계는 Irix조작체계하에서 처음으로 제안되였으며 2000년과 2001년에 SGI의 내부와 외부의 개발자그룹들에 의하여 Linux에 이식되였다. XFS는 Irix5.3조작체계용의 비실행기록EFS파일체계를 위한 교체판으로서 실리콘 그라픽스(Silicon Graphics)에 의하여 도입되였다.

앞으로 기억장치가 더욱 확대될것과 자료에 더 빨리, 더 효과적으로, 보다 안전하게 접근하고 봉사할수 있는 강력한 봉사기들이 필요할것이라는것을 예견하여 이에 대한 연구사업이 활발히 진행되였다. XFS의 설계와 개발방향을 요약하면 다음과 같다.

▼ 파일체계는 과학적인 파일과 콤퓨터봉사기로, 상업적자료처리봉사기로 그리고 수자식매체봉사기로 리용할수 있어야 한다.

- XFS는 오유로부터 신속히 복구되여야 하며 장시간 일관성 있게 디스크에 기초한 자료를 유지하여 응용성을 더욱 넓혀야 한다.
- 64bit규모의 파일에 대한 효과적지원능력을 가져야 한다. 서로 다른 대역을 가진 파일들이 블로크에로의 접근에서 성능상 위반이 아주 적거나 없어야 한다.
  일부 디스크공간위반(실례로 첨수들)으로 하여 성능을 더욱 높이게 한다. 대규모 파일의 끝에 있는 블로크까지의 찾기를 파일체계의 자료구조를 통하여 진행하는 선형탐색방법은 그다지 좋은것이 못된다.
- 예비파일에 대한 효과적지원능력이 있어야 한다. 우연적인 구멍(holes)처리도 지원되여야 한다. 구멍은 써지지 않고 령으로 읽어 지는 파일대역을 말한다. 또한 변경된 자료에 대한 검색과 새로운 자료의 삽입에서 표현이 디스크공간상으로나 CPU시간상으로도 효과적이여야 한다. 령으로 씌여 진 블로크(구멍에 의하여 교체될 목적으로 씌여 진)에 대하여서는 검사할 필요가 없다.
- 1KB정도로 작은 파일에 대하여서도 효과적으로 지원해야 한다. 표준뿌리(root)나 usr파일체계는 프로그람원천을 포함하는 파일체계와 마찬가지로 많은 파일을 가지고 있다. 대다수의 기호적련결들도 역시 이 항목에 일치한다.
- 대규모등록부들에 대하여 효과적인 지원기능 즉 탐색, 삽입, 제거와 같은 기능을 가져야 한다. 이것은 긴 등록부를 통한 선형탐색을 피하기 위하여 몇가지 종류의 첨수도식이 필요하다는것을 말해 준다. 고장으로부터 회복시간은 파일체계의 크기에 따라 증가하지 않는다. 그 시간은 고장상태의 어느 한 시각에 파일체계가 어떤 동작상태준위에 있는가에 따라 커질수 있다. 복구에서는 일관성을 확인하기 위하여 모든 색인마디와 등록부를 주사해야 한다. 이것은 선택하는데 따라(MS-DOS에서와 같이 동기적성질) 일관성은 가동일지에 의하여 담보되지만 속도가 상당히 떠진다는것을 암시해 준다. 사용자에게 성과적으로 귀환된 후에도 결속된 변경조작이 완료될 때까지 회복동작을 계속 진행한다. 일부 조작들은 가동일지

등록과 관련되는 조건에서는 최소한 동기화되여야 한다. 이것은 반드시 파일의 생성과 소거를 포함하며 부차적인 쓰기(완충화된)는 포함하지 않는다.

- ACL과 다른 POSIX1003.6을 기능적으로 지원해야 한다. 여기에는 몇가지 형태의 위임접근조종, 정보의 표식화, 듣기 등이 포함된다.

▲ 디스크의 섹터크기로부터 64KB 혹은 256KB까지의 범위에서 다른 론리적블로크 크기를 지원해야 한다. 블로크의 크기는 파일생성시에 설정된다. 이 크기는 파일체계에서 최소배정단위이다(색인마디를 제외하고).

최초의 개발자들은 이러한 목적을 념두에 두고 SGI의 프로그람작성자들과 기타 연구자들의 방조밑에 속도가 현저히 빠르며 효과적일뿐아니라 Linux2.4.x에 이식할수 있는 완전히 확장가능한 파일체계를 도입하였다. 이 책을 쓸 때에 XFS는 이미 변경된 2.2.1x와 보다 새로운 2.4.x를 다같이 안정베타판본에서 실행되고 있었다. 판본 2.4.4에서와 같이 XFS는 이미 표준Linux핵심부원천코드의 한부분으로 되였고 핵심부배치구성시의 선택항목의 하나로 되였다.

XFS의 실현

Linux체계의 다른 모든 파일체계와 마찬가지로 XFS는 VFS에 기초하여 실현되며 따라서 VFS의 장에서 론의된 모든 조작들이 XFS에도 그대로 적용된다.

XFS파일체계는 실행기록파일체계이다. 이 사실은 파일체계의 메타자료에 대한 갱신(색인마디, 등록부, 비트매프 등)이 최초의 디스크블로크를 갱신하기전에 디스크상의 직결가동일지구역에 기록된다는것을 의미한다. 조작이 실행되지 않거나 혹은 재실행되는것과 같은 폭주사건시에 가동일지에 주어 진 자료를 리용하여 파일체계를 안정한 상태로 회복시킬수 있다. 이와 같은 기술적실현은 체계가 폭주될 때 능동상태의 파일체계를 올려태우기전에 파일체계검사 및 회복프로그람(fsck)을 교체시키는것이다.

XFS는 이 책의 LVM장에서 서술된 LVM을 리용한다. LVM의 웃준위에서 XFS를 리용하려는 독자들을 위하여 현재 XFSCVS나무가(2001년 5월 현재로서의 XFS 1.0시험판) 이미 LVM beta6코드와 몇가지 XFS용의 묘안들을 포함하고 있다는것을 지적해 둔다. beta7에는 일부 론리적문제점들이 존재하기때문에 현행LVM판본(beta7)대신에 이 판본을 계속 쓸것을 권고하고 있다. XFS파일체계를 포함하는 LVM의 속성올려태우기동작은 파일체계가 실행기록속성이므로 진행될수 없다.

XFS는 64bit파일체계이다. 32bit응용프로그람으로부터 64bit파일에로의 접근을 지원하기 위하여 새로운 대면부가 64bit의 파라메터를 취할수 있게 정의되여야 한다. 이 대면부들은 응용프로그람이 체계호출준위아래로 64bit필터링(filtering)하는가 32bit필터링하는가에는 관계없이 명백하게 핵심부에서 지원되여야 한다. 파일체계와 관련한 호출은 모든 파일체계형태에 맞게 읽기, 쓰기, 열기, ioct1등으로 실현된다. 이 조작들은 VFS대면부를 통해 매개 파일체계형식에 관하여 서로 다른 부분프로그람들로 벡토르화되여 있다. 32bit와 64bit대면부는 다같이 기본 OS와 장치에 의하여 지원된다. 232bit보다 긴 파일에 대하여

고유하게 동작하는 32bit응용프로그람의 의미론적구조는 이 장의 마지막에서 취급한다. XFS의 기본 구성성분은 목록화되여 있다.

▼ 가동일지관리기-디스크공간의 개개의 대역에 대한 모든 메타자료변경을 직렬로 가동일지에 기록한다.
- 캐쉬관리기
- 잠금관리기-파일체계안의 디스크공간배정을 관리한다.
- 속성관리기-파일체계속성조작을 관리한다.
- 체계호출과 VFS대면부

▲ 이름공간관리기-경로이름을 파일참조로 변환하는 파일체계이름짓기조작을 실현한다.

가동일지관리기

파일체계메타자료에 관한 모든 변경내용(색인마디,등록부,비트매프 등)은 디스크공간의 하나의 분리대역에 직렬로 가동일지에 기록된다. 이것들이 매개 파일체계에 관한 개별적가동일지이다. 가동일지는 메타자료가 디스크에 기록되기전에 폭주장애가 발생할 때 일관성 있고 정확한 파일체계(회복)를 빨리 재구성할수 있게 한다. 가동일지공간은 안전을 위하여 파일체계공간으로부터 독립적으로 배정된다. 가동일지관리기는 캐쉬로부터 쓰기조작순서를 조종하는 공간관리기에 의하여 제공되는 정보들을 리용한다. 왜냐하면 특정한 가동일지는 폭주가 생길 때 정확성을 보장하기 위한 자료조작에 앞서서 혹은 차후에 순서화되여야 하기때문이다.

공간과 이름관리기부분체계는 가동일지관리기에게 가동일지등록요청을 보낸다. 매개 요청은 가동일지용의 특별한 가동일지블로크라든가 혹은 다중블로크에 채워 진다. 가동일지는 쓰기가 마지막끝에 도달할 때 덧붙여 지는 순환대기목록으로 실현된다. 매개 가동일지입구점에는 가동일지렬번호가 붙여 져 있으므로 가동일지의 끝은 제일 높은 번호를 조사하여 알아 낼수 있다.

폭주가 발생한후 가동일지는 파일체계가 리용되기전에 회복되여야 한다. 가동일지에 기록되고 완성된 조작들은 파일체계의 자료대역에는 아직 기억되지 않았기때문에 파일체계자료와 일관성상태를 정확하게 반영할수 있도록 재수행된다. 여기에서 가동일지관리기의 역할은 가동일지레코드들을 식별하며 회복조작을 수행하기 위하여 파일체계의 다른 프로그람부분을 호출하는것이다. 가동일지조작은 해당 기록권의 가동일지부분에 대하여 보다 높은 성능을 얻기 위하여 블로크들로 묶어 준다. 이 블로크를 가동일지레코드라고 부른다. 대표적으로 가동일지레코드들은 비동기적으로 기록되며 가동일지관리기는 체계의 보다 높은 준위에 될수 있는 한 빨리 현행가동일지레코드의 쓰기를 강하게 요구하도록 지령한다. 주어 진 가동일지의 쓰기는 선행한 가동일지가 완료될 때까지 출발하지 못한다.

캐쉬관리기

캐쉬는 기계적으로 마디국부화한 여러가지 파일체계의 디스크블로크의 캐쉬이다. 읽기요청은 캐쉬로부터 만족되며 쓰기요청은 캐쉬로 써질수 있다. 캐쉬입구점들은 사용빈도와 파일체계의 의미적구조를 고려하는 방향에서 새 입구점이 요구될 때 소거된다. 파일체계메타자료만이 아니라 파일자료도 캐쉬에 기억된다. 사용자요청은 기발설정에 의하여 (O_DIRECT)캐쉬를 쓰지 않을수도 있다. 그렇지 않은 경우에는 모든 파일체계 I/O들이 캐쉬를 거쳐 진행된다.

현재 완충기대면부는 두가지 방향으로 확장되고 있다. 첫째로 대면부의 64bit판본은 XFS의 64bit파일크기를 지원할수 있도록 첨가된다. 둘째로 캐쉬클라이언트가 조작기간에 완충기들을 수집하거나 변화된 완충기를 가동일지관리기에게 보내며 또 성과적으로 가동일지등록을 수행한 다음에 모든 완충기들을 개방할수 있게 거래기술이 제공된다. 앞으로의 배포판들에서는 캐쉬가 파일체계에 관하여 기계와 원격으로 자료를 보존하게 될것이다.

잠금관리기

잠금관리기는 잠금시간의 길이를 단축하고 보존하기 위하여 보다 개선된 알고리듬을 리용한다. 동시에 가동일지관리기설계에서는 파일체계의 객체들이 한개 클라스터만이 여러 마디안에서 잠금될수 있게 분산파일체계조작을 준비하고 있다.

공간관리기

공간관리기는 파일체계안에서 디스크공간배정을 관리하며 한개 파일(바이트렬)을 디스크블로크의 렬로 넘기기할 임무를 띠고 있다. 파일체계의 내부구조 즉 배정그룹(실린더), 색인마디, 빈 공간관리는 공간관리기에 의하여 조종되며 또 넘기기함수에 의하여서도 조종된다.

설계에서 공간의 배치구조선택은 64bit파일이 성기여 질 가능성을 포함하여 대규모파일이나 파일체계의 효과적지원요구의 영향을 받는다. 공간관리기는 또한 순차적처리를 진행할 때 찾기동작을 피할수 있게 블로크의 배치구조를 최량화할 임무를 가지고 있으며 디스크상에서 서로 가까이 있는 련관된 파일(같은 등록부안의)들을 유지해야 할 책임이 있다.

매개 파일체계는 가동일지, 메타자료, 자료 및 실시간부분기록권들로 분할된다. 보통 상태에서는 자료부분기록권과 실시간 부분기록권이 존재하지 않는다. 만일 자료부분기록권이 존재하면 보통 사용자자료는 그안에 존재하며 그렇지 않으면 메타자료는 부분기록권에 기억된다. 실시간파일을 위한 자료블로크들은 실시간부분기록권이 존재하면 그안에 기억되며 그렇지 않은 경우에는 메타자료부분기록권에 기억된다. 모든 파일체계자료들은 가동일지와 메타자료부분기록권에 기억된다.

공간관리기는 매개 파일체계메타자료와 자료부분기록권을 많은 배정그룹들로 분할한다. 부분기록권이 존재할 때 배정그룹은 메타자료와 자료부분기록권들로 구성되는 블로크들을 포함한다. 매개 배정그룹은 색인마디들의 모임과 자료블로크 그리고 배정을 조종하기 위한 자료구조체들을 가지고 있다. 색인마디를 포함하는 블로크들은 디스크공간을 더 효과적으로 리용하기 위하여 자료블로크들로부터 동적으로 배정된다. 배정그룹들을 위한 색인마디렬의 위치에 관한 정보는 보통파일(B나무에 있는)에 관하여서와 같은 방법을 취한다. 배정그룹안의 자유블로크들은 하나 혹은 두가지 도식에 의하여 계속 유지되게 된다. 첫 도식은 비트매프와 출발비트매프블로크에 의하여 구성되는 계수기들의 모임 그리고 빈 범위의 log2크기를 리용한다. 두번째 도식은 B나무의 쌍을 리용하는데 하나는 빈 범위의 크기에 의하여 첨수화되여 있고 다른 하나는 빈 범위의 출발블로크에 의하여 첨수화되여 있다. 사용될 도식은 두개의 조작들이 시험적으로 실현되고 그것들의 성능이 확인된후에 선택할수 있다. 실시간 부분기록권은 여러개의 고정크기의 범위들로 분할된다. 크기는 mkfs실행시에 선정하는데 적어도 1MB정도로 클것이 예견된다. 이 크기는 2의 루승으로 되지 말아야 하며 반드시 다중파일체계블로크크기로 되여야 한다. 이것은 기록권의 배치구성이나 그것의 다중화에서 리상적인 I/O크기로 될수 있다. 메타자료부분기록권에서 단일범위는 실시간부분기록권의 범위들을 위한 배정비트매프를 포함하며 다른 범위는 매 비트매프블로크(빈 범위의 번호)당 요약정보를 포함한다. 이 가변적인 배정방법은 실시간부분기록권이 객체로 선택되는데 그 근거는 고정크기범위에 의하여 배정이 가능하게끔 성능이 개선된데 있다.

파일의 기억은 파일의 크기와 접근에 의존하는 세가지 방법들가운데서 어느 하나로 표현된다. 작은 파일에서는 파일안의 자료가 색인마디에 기억된다. 중간규모의 파일에서는 색인마디가 파일의 자료를 가지고 있는 크기에 대한 지적자를 포함한다. 대규모의 파일에 관하여서는 색인마디가 파일안의 론리적위치에 의하여 첨수화된 B나무의 뿌리블로크를 포함하는데 뿌리블로크에서 레코드들은 파일자료를 포함하는 디스크범위들을 지적한다. 이 기억구조는 대규모적으로 토막화되고 성긴파일을 B나무첨수관리로 약간한 휴지시간을 가지고 효과적으로 실현되게 한다.

능동파일체계는 보다 많은 공간을 토대기록권에 보충하는 방법으로 확장할수 있다. 이 조작은 공간관리기에 의하여 직결방식으로 지원되는데 이때 공간관리기는 파일체계확장에 대한 요청을 접수하고 그것을 실현하기 위하여 디스크상의 구조와 기억내구조를 갱신한다. 앞으로의 실현에서는 단일파일체계에서 공간관리조종을 체계의 다중마디환경으로 분할하는 기능을 지원할것이 요구된다. 첫번째 실현에서는 이 기능을 고려하지 않은것이다.

속성관리기

속성관리기는 파일체계의 속성조작을 실현한다. 즉 이름공간에서 객체와 련관된 임의의 사용자정의속성에 대한 검색과 기억조작을 실현한다. 속성이란 이름과 값의 쌍을 의미하는데 여기서 이름은 인쇄가능한 문자렬이며 값은 임의의 작은 바이트렬이다. 속성을 사용자응용프로그람이나 혹은 핵심부에 의하여 조종할수 있다. 어떤 속성들은 체계에

의하여 미리 정의되며 UNIX대면부와 새로운 속성접근체계호출 실례로 파일접근과 변경시간 등을 리용하여 접근할수 있다.

속성은 참조되는 객체의 색인마디에 첨가하는 방법으로 내부적으로 기억되나 속성관리기는 색인마디와 련관된 속성구조체를 관리한다. 그러나 파일의 허가권과 시간표식과 같은 이름공간관리기에 의하여 조종되는 마당들은 관리하지 않는다.

객체가 생성될 때 배정되는 임의의 속성은 하나도 없으며 존재하는 임의의 속성은 객체가 파괴될 때 역시 파괴된다. 속성은 색인마디들사이에 서로 공유되지 않는다.

접근조종목록들은 색인마디들사이에서 속성이 공유되는것과 같은 특별한 경우로서 조종된다. 이것은 임의의 속성이나 혹은 속성값을 가진 파일체계의 모든 객체들을 빨리 찾을수 있다는것을 의미한다. 일부 응용체계프로그람들은 속성에 대하여 알아 내기 위하여 변경될수도 있는데 례를 들어 cp는 파일을 복사할 때 선택된 속성들도 복사한다. 체계여벌복사편의프로그람은 객체를 여벌복사하거나 회복할 때 객체의 속성도 여벌복사하고 회복한다. 표준NFS는 전통적인 UNIX모임을 초월하여 속성들을 지원하지 않는다. 따라서 이 속성들은 표준 NFS를 통하여 XFS파일체계로 접근하는 클라이언트에 대하여서는 어떤 방법으로도 볼수 없다.

이름공간관리기

이름공간관리기는 파일체계의 이름짓기조작들을 실현하며 경로이름을 파일참조로 변환한다. 파일은 내부적으로 파일체계와 색인마디번호에 의하여 식별된다. 색인마디는 파일에 관한 정보를 유지하는 디스크상의 구조체이다. 색인마디번호는 특정한 파일체계안의 색인마디의 표식(혹은 첨수)이다. 파일은 또한 파일유일id라고 부르는 파일에 관한 유일한 수치적값에 의하여 식별된다. 파일체계는 "magic cookie", 대표적으로 뿌리색인마디의 기억주소라든가 혹은 파일체계유일id에 의하여 식별된다. 파일체계유일id는 파일체계가 생성될 때 그리고 파일체계가 소거될 때까지 그 파일체계와 유일적으로 련관될 때 지적된다. 보충적인 림시적유일id 즉 파일체계 I/O유일id는 파일체계가 올려태우기될 때마다 생성되며 올려태우기기간에만 유효하다.

이름공간관리기는 등록부구조체와 파일허가권이나 시간표식과 같은 공간관리에 관련없는 색인마디의 내용을 관리한다. NFS를 비롯하여 다른 파일체계로부터 제기되는 요청은 이름공간관리기가 색인마디를 찾는데 리용하는 파일핸들을 가진 체계에 도달한다. 이 파일핸들은 파일체계나 색인마디를 추론하는데 충분한 정보를 가지고 있으며 파일체계의 판본도 표시한다. 분산XFS파일체계에서 다른 마디들로부터 오는 요청은 파일체계의 유일id나 색인마디번호, 파일유일id로 들어 갈수 있으며 이 시점에서 두가지 식별형이 맞는다는것을 립증할수 있다. 이름공간관리기는 이름짓기조작을 고속으로 진행하기 위하여 캐쉬를 리용할수도 있다. 이름변환과 관련한 구체적내용은 호출자로부터 숨겨 져 있다.

현재 이름공간관리기의 올려태우기의미론의 설계에는 올려태우기점이라고 부르는 파일체계마디가 있는데 이 올려태우기점은 현행파일체계의 빈등록부올려태우기점을 교체하는데 리용된다. 올려태우기점마디는 파일체계유일id를 포함한다. 이름짓기조작기

간에 원격파일체계를 참조하는 올려태우기점을 만나게 되면 통보문이 이름짓기요청에 따라 조작을 완성하기 위하여 그 파일체계id를 가진 파일체계를 관리하는 원격기계에 전송된다.

선택적이며 확장성 있는 이름짓기도식은 통보문대기렬의 다른 한끝에 존재하는 실체를 프로그람화함으로써 사용자방식으로 실현될수 있다. 이 방식은 체계의 첫번째 발표에서는 실현되지 않은 내용이다.

XFS파일체계의 관리

XFS관리는 기록권과 파일체계를 생성하거나 유지하는데 필요한 편의프로그람들을 포함한다. 또한 기록권조종, 파일체계조종, 올려태우기, 올려태우기해제, 여벌복사, 회복, 계층적파일체계 등을 위한 프로그람적인 대면부를 포함한다. 앞으로 관리지원기능은 기록권과 파일체계의 올려태우기, 여벌복사 기타 기능에 대한 원격접근을 가능하게 할수 있도록 확장될것이다. 그라픽스대면부들은 필요한 새로운 도구 즉 기록권관리를 위한 MSD의 체계관리그룹에 의하여 제공될것이다.

XFS구조체와 조작

이 절에서는 XFS의 핵심안의 색인마디의 생성, 관리, 소거에 대하여 서술한다.

색인마디의 자료구조체

XFS의 핵심안의 색인마디구조체는 다음과 같이 정의된다.

```
typedef struct xfs_inode {
struct xfs_ihash *i_hash; /* pointer to hash header */
struct xfs_inode *i_next; /* inode hash link forw */
struct xfs_inode **i_prevp; /* ptr to prev i_next */
struct xfs_mount *i_mount; /* fs mount struct ptr */
struct xfs_inode *i_mnext; /* next inode in mount 's list */
struct xfs_inode **i_mprevp; /* ptr to prev i_mnext */
struct vnode *i_vnode; /* ptr to associated vnode */
dev_t i_dev; /* dev containing this inode */
xfs_ino_t i_ino; /* inode number (agno/agino) */
xfs_agblock_t i_bno; /* ag block # of inode */
int i_index ; /* which inode in block */
xfs_trans_t *i_transp; /*ptr to owning transaction */
```

```
xfs_inode_log_item_t i_item; /* logging information */
mrlook_t i_lock; /* inode lock */
sema_t i_flock; /* inode flush lock */
unsigned int i_pincount; /* # of times inode is pinned */
sema_t i_pinsema; /* inode pin sema */
ushort i_flags; /* misc state */
ulong i_vcode; /* version code token (RFS) */
ulong i_mapcnt; /* count of mapped pages */
ulong i_update_core; /* inode timestamp ditry flag */
size_t i_bytes; /* bytes in i_ul */
union {xfs_bmbt_rec_t *iu_extents; /* linear map of file extents */
char * iu_data; /* inline file data */}
i_ul; xfs_btree_block_t *i_broot; /* file's in-core b-tree root */
size_t i_broot_bytes; /* bytes allocated for root */
union {xfs_bmbt_rec_t iu_inline_ext[2]; /* very small file extents */
char iu_inline_data[32]; /* very small file data */
dev_t iu_rdev; /*dev number if special */
xfs_unid_t iu_unid; /* mount point value */ }
i_u2; ushort i_abytes; /* bytes in i_u3 */
union {xfs_bmbt_rec_t *iu_aextents; /* map of atribute extents */
char*iu_adata; /* inline atribute data */}
i_u3; xs_dinode_core_t i_d; /* most of the on-disk inode */}
xfs_inode_t;
```

색인마디의 생명주기

이제 우리는 디스크를 읽는 시각부터 핵심안의 색인마디구조체가 핵심부의 히프(heap)에 귀환되는 시간까지의 핵심안의 색인마디생명주기를 고찰한다.

1 단계(*핵심안의 색인마디배정*)

우선 핵심안 색인마디사용자는 xfs_iget() 혹은 xfs_trans_iget()을 호출하여 색인마디를 취하도록 한다. 호출자는 요구되는 색인마디의 색인마디번호를 정의하며 색인마디가 독점방식으로 잠금되였는지 혹은 공유되였는지를 정의하며 함수는 초기화된 핵심안 색인마디에 대한 지적자를 귀환시킨다. 이 조작은 보통 파일부분에 대한 검색으로 실현된다. 색인마디는 증가된 색인마디의 vnode참조계수값으로 잠금되며 귀환된다. 다른 것들은 동일한 색인마디에 대한 참조값을 가질수 있으며 색인마디잠금은 구조체에 대한 접근을 동기화하는데 리용된다.

2단계(*색인마디의 찾기*)

프로쎄스가 색인마디를 일단 포착하면 그것을 찾는다. 만일 프로쎄스가 색인마디를 변경시키려고 하면 색인마디는 독점적으로 잠그어 진다. 그러나 만일 색인마디를 읽기만 하면 색인마디는 공유된 상태에서 잠그어 진다. 실제적으로 색인마디들은 보통 lookupname(), 부분프로그람에 의하여 탐색되기때문에 사용자는 보통 한개의 참조값을 가지고 inode/vnode를 얻지만 잠금은 하지 않는다. 다음 사용자는 xfs_ilock()를 호출하여 색인마디를 명백하게 잠근다. 일단 색인마디가 잠금되면 그것을 유지하고 있는 프로쎄스는 임의의 마당으로부터 읽을수 있다. 하지만 색인마디가 거래의 문맥으로 될 때는 변경만 할수 있다.

3단계(*색인마디의 변경*)

색인마디가 변경되려면 독점적으로 잠그어 져 있어야 한다. 호출자가 거래지적자를 취하는것을 제외하고는 xfs_iget()와 꼭같은 cfs_trans_iget()를 호출하여 색인마디를 얻지 않으면 xfs_ilock()의 호출에 의하여 독점적으로 잠그어 지며 xfs_trans_ijoin()이 호출에 따라 거래에 첨부된다. 이 조작이 수행되기만 하면 색인마디를 변경할수 있다. 색인마디에 대한 모든 변화가 이루어 지면 거래공정은 색인마디내에서 무엇이 변경되였는가를 통보해야 하며 따라서 가동일지에 기록되여야 한다.

4단계(*색인마디변경에 대한 가동일지등록*)

색인마디에 대한 변경은 xfs_trans_log_in_ode()함수를 리용하는 색인마디의 한 부분으로서 가동일지에 기록될수 있다. 이 함수는 색인마디의 부분들이 변경되였다는것을 지적하는 기발들을 얻고 가동일지에 기록할 필요가 있다. 기발들은 모두 xfs_inode_item_h안에 정의되며 이 장의 다음 부분에서 구체적으로 서술될것이다.

5단계(*색인마디의 개방*)

일단 프로쎄스가 색인마디를 찾고 변경시키면 그 색인마디는 잠금을 해제하여야 하며 개방되여야 한다. 만일 색인마디가 거래의 부분으로 리용되지 않았으면 프로쎄스는 색인마디의 잠금을 해제할 xfs_iput()를 쉽게 호출할수 있으며 색인마디의 vnode에 대한 참조값을 개방한다. 만일 색인마디가 거래의 부분으로 사용되고 있으면 프로쎄스가 xfs_trans_commit()를 호출할 때 색인마디는 잠금해제되며 그것의 참조값도 해제된다. 만일 프로쎄스가 결속을 완료한후에도 색인마디우에 계속 유지되려고 하면 거래를 결속하기전에 xfs_trans_ihold()를 호출할 필요가 있다. 이것은 거래코드는 거래가 결속될 때에는 색인마디의 잠금을 해제하거나 개방하지 않는다는것을 말한다.

6단계(*색인마디변경의 재쓰기*)

색인마디가 거래의 한 부분으로 변경될 때 변경된 색인마디구조체는 기억상태에 있으며 그 변경내용은 디스크상의 가동일지에 기록된다. 어떤 점에서 색인마디는 xfs_sync()

를 호출하는 bdflush()에 의하여 혹은 다른 색인마디로써 사용할것을 요구하는 색인마디구조체에 의하여 디스크상의 홈(home)에 재쓰기될수 있다. 이 경우에 색인마디에 대한 모든 변경들은 이 점에서 디스크에 재쓰기되며 디스크상의 가동일지복사는 파일체계회복에 더이상 필요없게 된다.

7단계(*색인마디구조체의 해제*)

우에서 언급한바와 같이 색인마디구조체는 어떤 점에서 다른 색인마디를 리용하기 위하여 재생될수도 있다. 이 경우에는 색인마디의 기억이 개방되고 그 어떤 다른것에 재리용되므로 핵심안 색인마디의 생명주기가 끝나게 된다.

색인마디의 배정

핵심안 색인마디들은 xfs()_iget()의 호출에 의하여 배정된다. xfs_iget()의 함수원형은 다음과 같다.

xfs_inode_t*xfs_iget(xfs_mount_t*mp, xfs_trans_t*tp, xfs_ino−tino, unit flags)

호출자는 파일체계의 올려태우기구조체에 대한 지적자와 만일 거래를 실행하면 거래지적자를 주며 요구되는 색인마디의 색인마디번호 그리고 색인마디가 공유방식에서 잠금되였는지 독점방식에서 잠금되였는지를 지적하는 기발들을 준다. 함수는 요구한 색인마디의 핵심안 판본에 대한 지적자를 귀환시킨다. 색인마디는 요청된 방식에서 잠금된 호출자에게 귀환된다. 색인마디의 마당들은 디스크상의 색인마디의 양식에 따라 채워 지게 된다. 만일 색인마디파일자료가 디스크상의 색인마디안에 전체적으로 일치한다는것을 의미하는 LOCAL양식으로 되여 있으면 i_u1. iu_data가 파일내용을 가지고 있는 기억내배렬을 지적하게 되며 i_byte는 배렬안의 바이트번호를 포함하게 된다. 이 배렬은 파일자료가 32byte와 같거나 작으면 i_u2. iu_inline_dataarray로 되거나 혹은 핵심부히프로부터 배정된 배렬일수 있다. 만일 파일의 길이가 0이면 i_u1. iu_data는 NULL로 되며 i_byte는 0으로 된다. 또한 색인마디가 EXTENTS양식이면(파일의 자료가 디스크상의 색인마디와 일치되지 않지만 색인마디에 대한 범위서술자이라는것을 의미하는) i_u1. iu_extent는 파일의 범위서술자를 포함하는 기억내배렬을 지적할것이며 i_byte는 배렬안의 바이트번호를 포함할것이다. 이 배렬은 한개 혹은 두개의 범위만 있으면 i_u1. iu_inline_ext array일수 있거나 혹은 핵심부의 히프로부터 배정된 배렬일수 있다. 파일의 길이가 0이면 i_u1. iu_extent는 NULL로 되며 i_byte는 0으로 될것이다. XFS_IEXTENTS기발은 i_flags에 설정된다. 이 기발은 파일의 모든 범위서술자를 읽을수 있으며 i_u1. iu_extents배렬안에 있다는것을 지적한다.

만일 색인마디가 파일이 디스크상의 색인마디에 일치시키기 위한 범위서술자가 너무 많다는것을 의미하는 B나무양식이면 i_broot는 파일의 범위 B나무뿌리를 포함하는 기억내배렬을 지적할것이며 i_broot_byte는 배렬안의 바이트수를 포함할것이다. 또한 XFS_IEXTENTS기발이 i_flags에 설정되면 i_u1. iu_extents는 우의 경우에서와 같이 파일의

모든 범위들을 포함하는 배렬을 지적할것이다. 만일 기발이 령으로 설정되면 범위는 여전히 읽을수 없으며 필요하다면 xfs_ireadindir()로 호출하여 읽어야 한다. 큰 범위들을 가지는 파일의 모든 범위들에 대한 읽기는 간단히 파일의 stat()로 효과적으로 수행될수 있다. 귀환된 색인마디는 매 파일당으로 핵심안 색인마디하쉬표로 하쉬화된다. 우리는 모든 파일체계를 위한 전통적인 단일하쉬표로부터 체계의 유연성을 개선하기 위하여 매 파일당 하쉬표로 방향을 바꾸었다. xfs_iget()의 호출은 우선 디스크로부터 색인마디를 옮겨쓰기전에 이 하쉬표에서 요구되는 색인마디를 찾는다. 핵심안 색인마디는 오직 재생될때만 하쉬표로부터 제거된다. 색인마디는 또한 파일체계의 올려태우기구조체에 첨부된 목록우에 배치될수도 있다. 이 목록은 xfs_sync()와 같은 부분프로그람에서 모든 핵심안 색인마디를 추적하는데 리용된다.

색인마디의 직접삽입자료/범위/B나무뿌리

이 부분은 iu_data, iu_extents, i_broot마당들의 조종을 서술한다. 이 마당들은 범위가 파일의 크기변화만큼 변화되여야 할 배렬을 지적한다.

iu-data

색인마디배정에 대한 부분에서 서술된바와 같이 이 마당은 LOCAL양식으로 색인마디의 직결자료를 포함하는 배렬을 지적한다. 이 양식의 파일크기가 변화될 때 크기를 변경시키는 프로쎄스는 핵심안 배렬의 크기를 재설정하기 위한 xfs_idata_realloc()를 호출해야 한다. 이 함수는 필요되는 바이트수로 델타(delta)를 취한다. 만일 델타가 정이면 배렬에 더 많은 기억이 배정되며 부이면 더 작아 진다. 만일 크기가 령으로 되면 iu_data는 NULL로 된다. 만일 크기가 디스크상의 색인마디에 일치되는것보다 더 커지게 되면 프로쎄스는 LOCAL양식으로부터 EXTENTS양식으로 변화시키기 위하여 거래를 수행하도록 크기를 변경시켜야 한다. 그러한 거래의 한 부분으로서 직결자료가 가동일지에 기록되여야 하며 iu_data배렬은 xfs_idata_realloc()를 호출하여 개방되여야 한다. 그리고 색인마디의 범위를 위한 배렬은 xfs_iext_realloc()을 호출하여 iu_extentes에 배정되여야 한다.

iu_extents

색인마디가 EXTENTS나 BTREE양식에 있으면 이 마당은 그 색인마디에 관한 모든 범위서술자들을 포함하는 배렬을 지적한다. 만일 파일이 EXTENTS양식에 있으면 이 배렬은 파일의 길이가 0이 아닌 이상 존재하는것으로 담보된다. 만일 파일이 BTREE양식이면 이 배렬은 첫번째로 요구될수도 있으며 그의 존재는 색인마디의 기발마당의 XFS_IEXTENTS에 의하여 표현된다. 파일에서 범위의 수가 변할 때 프로쎄스는 핵심안 배렬의 크기를 재설정하기 위하여 xfs_iext_realloc()를 호출해야 한다. 이 함수는 필요한 범위의 수에서 델타를 취하고 필요한 배정만큼 배렬의 기억을 개방한다. 범위의 수가 0으로 되면 iu_extents는 NULL로 설정된다. 만일 범위의 수가 아직 채 결정되지 않은 턱값을 초과하면 배렬은 더 커지지 않고 블로크넘기기코드는 캐쉬의 B나무를 통한 접근속

도가 더 떠지게 된다. iu_extents배렬은 파일변화에 따라 정렬된 색인마디의 모든 범위를 포함한다. 이 변위값은 파일디스크블로크의 위치를 빨리 찾기 위한 블로크넘기기코드에 리용된다. 이 조작은 직렬접근검색의 효과성을 개선하기 위하여 단일입구점캐쉬로 더욱 강화된 배렬의 2진탐색에 의하여 수행된다.

iu_broot

색인마디가 BTREE양식일 때 이 마당은 디스크상의 색인마디의 B나무뿌리의 핵심안복사를 지적한다. 우에서 언급된 다른 배렬과 같이 이 배렬은 사용된 B나무뿌리부분을 유지하기 위하여 충분한 기억을 확보하며 뿌리가 커지거나 줄어 드는것만큼 동적으로 크기를 다시 설정하여야 한다. 이 조작은 xfs_iroot_realloc()의 호출에 의하여 실현되는데 이 함수는 요구된 B나무레코드의 수를 변화시킨다. 이 부분프로그람은 B나무뿌리의 양식을 파악하며 크기를 변경시킬 때 적당한 값으로 뿌리안에 존재하는 정보를 움직인다. 뿌리의 크기가 일부 레코드에 의하여 증가할 때는 레코드에 대한 지적자들이 자료구조체의 끝을 향하여 옮겨 지며 크기가 줄어 들 때는 지적자들이 앞으로 옮겨 진다. B나무뿌리의 레코드수가 0으로 될 때 자료구조체에는 B나무블로크머리부에 그 값이 계속 남아 있기 때문에 개방되지 않는다. 더이상 필요없으면 프로쎄스는 뿌리를 포함하는데 리용된 기억을 개방하기 위하여 xfs_iroot_free()를 호출해야 한다. 이 과정은 색인마디가 더이상 BTREE양식에 있지 않을 때만 수행된다.

i_byte와 i_broot_bytes

이 두개의 계수기는 대응하는 iu_data/iu_extents와 i_broot마당들에 의하여 지적되는 배렬에 리용된 바이트들을 추적한다. 당분간 iu_inline_data/iu_ inline_ext배렬의 리용을 제외하고 배렬들은 정확히 요구되는 크기에 있다. 이것은 우리가 kmem_realloc()를 호출해야 하거나 혹은 xfs_i***_realloc()부분프로그람들중의 하나가 호출될 때마다 매번 그와 류사한 어떤 함수를 호출해야 한다는것을 말한다. 우리는 kmem의 호출수를 줄이려고 사용하는것보다 더 많은 기억을 유지하지만 현재 더 좋은 기억리용을 추구하여 상업화되고 있다. 이것이 cpu의 주기로 환산하여 고도의 휴지시간해결책으로 된다면 그것을 변경시킬수 있다.

색인마디잠금

우에서 언급한바와 같이 색인마디는 공유방식이나 혹은 독점방식으로 잠금을 실현할수 있다. 이것은 같은 색인마디에 대하여 동시에 여러명의 읽기를 수행할수 있다는것을 의미한다. 이때 동일한 색인마디는 다중읽기조작과 파일의 비배정쓰기조작을 병렬로 진행할수 있어야 한다. 동시파일접근은 비동기I/O와 등록부에 관하여 특별히 중요하다. 우리의 비동기I/O실현은 스레드에 기초하고 있으며 따라서 같은 시각에 여러개의 스레드가 파일에 접근하게 하는것은 크고 구획으로 분할된 기록권들과 같은 장치들의 성능을 높이기 위한 관흐름식I/O요청을 증가시킨다. 등록부들은 쓰는것보다 더 자주 경로탐색을 진행하여 읽을수 있으며 따라서 일반 장치들에 대한 병렬접근가능성으로 하여 경로결정성

능을 더욱 높일수 있게 한다. 이것은 다른 파일체계에서의 주목하는 《병목》문제로 되며 따라서 이것은 한걸음 더 전진한것으로 된다. 색인마디잠금은 색인마디내용갱신을 위하여 독점적으로 유지되여야 한다. 이때 색인마디의 내용은 디스크상의 색인마디에 포함된 모든 마당들을 포괄하며 필요하면 다른것들도 포함한다.

색인마디거래와 가동일지등록

디스크상의 색인마디에 반영될수 있는 색인마디의 모든 변경내용은 거래의 문맥안에서 이루어 져야 하며 그 거래안에 가동일지에 기록되여야 한다. 이와 관련하여 유일하게 제외되는것은 다음 부분에서 서술하게 될 색인마디상에서의 접근, 변경, 수정회수일수도 있다. 일단 색인마디가 수정되면 거래기법은 xfs_trans_log_inode()를 호출하여 그 변경을 알려야 한다. 이 함수는 색인마디부분들이 변화되였다는것을 지적하는 기발모임을 장악한다. 기발들에는 다음과 같은것들이 있다.

- ▼ XFS_ILOG_META:이 기발은 i_d부분구조체의 임의의 마당이 변경되면 정의되여야 한다.
- XFS_ILOG_DATA：이 기발은 색인마디의 직결자료가 변화되면 정의되여야 한다.
- XFS_ILOG_EXT:이 기발은 iu_extents배렬이 변경되고 파일이 EXTENTS양식이면 정의되여야 한다.
- XFS_ILOG_BROOT:이 기발은 파일이 BTREE양식이고 i_broot배렬의 내용이 변경되면 정의되여야 한다.
- ▲ XFS_ILOG_DEV:이 기발은 i_u2.iu_rdev마당이 변경되면 정의되여야 한다. 이 기발들은 현재 거래가 결속될 때 색인마디의 부분들이 가동일지에 기록되려고 한다는 것을 거래코드에 알려 준다. 매 정의부분은 입구점안의 가동일지에 기록되며 그리하여 XFS_ILOG_META의 정의는 핵심안 색인마디에 매몰된 전체 xfs_dinode_core구조체로 가동일지에 기록하며 XFS_ILOG_ BROOT의 정의는 전체 B나무뿌리를 가동일지에 기록하게 될것이다. 우리는 작은것 즉 색인마디의 작은 부분들에 대하여서는 가동일지에 기록하지 않는다. 왜냐하면 작은 부분들을 추적하는데 소비되는 휴지시간은 그것들을 가동일지에 복사하는것만큼 높아 지기때문이다.

색인마디를 조종하는 거래가 결속될 때 색인마디는 잠금해제되며 색인마디에 대한 참조는 개방된다.

색인마디소거

변경된 색인마디들은 bdflush데몬을 호출하는 방법으로 디스크에 대하여 소거되거나 색인마디의 가동일지이메지가 가동일지의 뒤쪽에 너무 멀리 있을 때 거래관리코드에 의하여서도 소거된다. 색인마디는 색인마디의 변경으로부터 다른 프로쎄스를 보호하기 위하여 공유방식으로 잠금되여야 하며 디스크에 기록되는 동안에 다른 프로쎄스가 색인마

디를 찾을수 있게 한다. 다중프로쎄스들은 색인마디들을 동시에 소거하려고 할수 있기때문에 i_flock는 색인마디의 소거를 표기화하는데 리용된다. 이 조작은 성능과 함께 정확성을 다같이 필요로 한다. 성능에 관하여서는 불필요한 작업을 할 필요가 없다. 정확성에 관하여서는 거래관리코드에 의하여 얻어 진 색인마디에 대한 참조값이 리용되지 않거나 혹은 다중프로쎄스에 의하여 개방되지 않는다는 사실을 확인해야 한다. 일단 참조값이 개방되면 색인마디가 재생될수 있다. 따라서 한개 프로쎄스만은 참조값이 색인마디를 보호할수 있다고 가정할수 있다.

색인마디의 실제적소거를 수행하는 부분프로그람은 xfs_iflush()이다. 이 부분프로그람은 캐쉬로부터 색인마디의 디스크블로크에 해당한 완충기를 얻을수 있게 하며 색인마디를 그 완충기에 복사하고 디스크에 완충기를 동기적 혹은 비동기적으로 혹은 일정한 지연을 가지고 재쓰기할수 있게 한다. 만일 색인마디가 기억에 꽉 붙잡혀 있으면(아직 디스크에 결속되지 않은 트랙잭션의 한부분이기때문에) 이 부분프로그람은 알려 질 때까지 대기하게 된다. 만일 색인마디가 붙잡혀 있지 않으면 그 부분프로그람은 xfs_iflush_done()과 완충기의 b_iodone함수에 대한 색인마디가동일지항목 그리고 b_fsprivate지적자를 붙인다. 이 부분프로그람은 Active Item List (AIL)쓰기가 완성될 때 실행되게 된다. 목적은 체계의 능동항목목록(AIL)으로부터 색인마디를 제거하며 색인마디지우기를 표시하며 거래코드에 의하여 얻어 진 색인마디에 대한 참조를 개방하는데 있다. 끝으로 색인마디는 지워 졌다는것과 잠금이 해제되였다는것 그리고 완충기쓰기가 시작되였다는것을 표시한다.

완충기가 xfs_iflush()로 잠금이 해제되면 쓰기를 완료하고 xfs_iflush_done()를 실행하기전에 색인마디가 다시 낡아 질수 있다. 이 경우에는 색인마디가 AIL의 앞쪽으로 옮겨진다. 한편 xfs_iflush_done()에 의하여 완료되는 소거는 AIL로부터 색인마디를 제거할 권한을 가지고 있지 못한다.

이것을 조종하기 위하여 xfs_iflush_done()은 다음과 같은 동작을 수행한다.

- ▼ 먼저 AIL의 잠금(이 마당을 보호하는)을 취하지 않고 색인마디의 LSN을 찾는다. 만일 그것이 변경되였으면 색인마디가 AIL에 옮겨 졌거나 옮겨 지고 있으며 이에 대하여 우려하지 않아도 된다.
- 만일 값이 변경되였으면 AIL의 잠금을 얻고 값이 아직 변경되지 않았으면 AIL로부터 색인마디를 제거한다.
- 다음 색인마디의 i_flock를 개방한다.
- ▲ 마지막으로 색인마디에 대한 참조값을 개방한다.

일단 참조값이 개방되면 더이상 조종할수 없거나 색인마디를 찾을수 없다. 여기서 디스크쓰기가 발생하는 동안 다시 색인마디를 지적하는 색인마디참조값으로 어떤 특별한 동작은 수행하지 않는다는데 대하여 강조해 둔다. 색인마디는 잠금을 해제하기전에 xfs_i_flush()로 지우기가 표기되기때문에 기록하는 동안에 색인마디를 변경시키는 임의의 프로쎄스는 거래코드에 관한 다른 참조값을 얻게 된다.

색인마디재생

어떤 시점에서 xfs_reclaim()의 호출은 참조되지 않는 색인마디를 재생시키려고 할것이다. 색인마디는 이 시점에서 참조가 없다는것을 담보하므로 색인마디가 낡아 지지 않았다는것을 알수 있다. 이제 해야 할 조작은 색인마디와 련관된 임의의 변경된 파일자료를 소거하며 핵심안 색인마디의 올려태우기구조체의 목록으로부터 색인마디를 제거하며 색인마디와 관련된 모든 기억을 개방하는것이다.

XFS상위블로크구조체와 조작

상위블로크는 대다수의 거래에 의하여 변경되는 중심적인 자원이다. 이것은 상위블로크가 병목현상에 대하여 높은 잠재력을 가진다는것을 의미한다. 이것은 일단 거래가 자원을 변경하면 그 자원은 처음으로 결속될 때까지 다른 거래에 보여 질수 없기때문이다. 상위블로크와 같은 자원이 많은 거래에 의하여 접근되고 매개가 일정한 시간동안 자원을 유지하면 거래는 자원의 접근을 기다리면서 병목화될것이다. 이러한 현상을 방지하기 위하여 XFS에서 상위블로크는 잠금을 유지하는 시간을 최소화할수 있게 설계된 부분프로그람들을 통하여 변경되게 될것이다. 상위블로크가 변경되는 리유는 상위블로크가 파일체계의 전체 색인마디의 수, 자유색인마디의 총수, 자유블로크의 총수를 포함하고 있기때문이다. 이 계수기들은 대다수시간 변경되며 따라서 최량화되는 값들로 갱신되게 된다. 공통적인 경우에 대하여 최량화의 일반규칙에 관하여 일관하지만 상위블로크에 대한 갱신이 완전히 공통이면 상위블로크의 다른 마당들은 거래안에서 때때로 변경될것을 요구하게 된다. 이러한 비공통적경로에 대한 대면부는 어떠한 객체도 중단시킴이 없이 이 마당들을 가지고 작업해야 한다. 이 모든 내용은 역시 상위블로크의 핵심안복사에 대한 접근과 대응되여야 한다.

상위블로크완충기

표준getblk()/bread()경로를 거쳐 접근되는 공통적인 캐쉬에 보존하지 않고 XFS상위블로크는 파일체계에 대하여 전통화한 완충기에 보존되게 된다. 캐쉬코드가 bdflush()에서 수행한것과 같은 이 방법은 상위블로크완충기와 전혀 호상작용하지 않는다. 물론 이것은 상위블로크가 요구될 때마다 디스크에 대하여 소거된다는것을 확인하여야 한다는것을 의미한다. xfs_sync()부분프로그람이 주기적으로 호출되기때문에 필요할 때 그로부터 상위블로크를 소거하는것이 보다 효과적이다.

상위블로크완충기는 올려태우기구조체에 보존된 지적자에 의하여 쉽게 지적할수 있다. 완충기에 대한 접근은 아래에서 구체적으로 서술되는 xfs_getsb()와 xfs_trans_getsb()부분프로그람을 통하여 진행된다. 이 부분프로그람들은 상위블로크완충기에 대한 접근을 동기화할 목적을 가지고 있다. 완충기는 오직 올려태우기시에 디스크로부터만 읽어 지게 되며 그 다음 완충기는 상위블로크의 디스크우복사로 sync에 보존되게 된다. 이것은 상위블로크에 대한 접근이 I/O동작 등으로 지연되지 않는다는것을 담보해야 한다. 왜냐하면

다른 지원들도 완충기에 재생을 강하게 요구하고 있기때문이다.

상위블로크완충기가 거래기간에 잠금을 유지하는 시간을 최소화하기 위하여 상위블로크는 실제적으로 거래가 결속되기전에는 잠금되거나 변경되지 않는다. 이것은 디스크로부터 다른 자원들을 읽어 들이거나 다른 거래에 의하여 개방되기를 기다리는 동안에 완충기가 잠금을 유지하지 않는다는것을 담보한것이다. 갱신과정이 절대수값보다도 계수기에 더해지거나 그로부터 떨어 지는 값을 포함하기때문에 공통적인 경우에 이 조작은 계수기에 관하여 잘 진행된다. 이 갱신값들은 거래의 정확성을 보장하면서 끝까지 지연될수 있다. 거래의 사용자는 이전 변화가 상위블로크에 xfs_trans_mod_sb()의 호출을 적용할 필요가 있다는것을 지적하게 된다. 이 부분프로그람은 요청된 변화를 기록하며 거래의 부분으로서 상위블로크에 적용된다는것을 보증해야 할 책임을 지고 있다.

계수기와 다른 상위블로크의 마당들이 변경되여야 할 때 상위블로크완충기는 xfs_trans_getblk()/xfs_trans_bread()보다는 오히려 완충기가 xfs_trans_getb()의 호출로 얻어져야 한다는것을 제외하고는 임의의 다른 완충기로서도 접근될수 있다. 상위블로크완충기를 리용하는 xfs_trans_log_buf()에 대한 호출은 잘 진행되며 xfs_trans_mod_sb()의 리용과 혼합될수도 있다.

핵심안 상위블로크는 파일체계에 대한 정적정보만이 아니라 체계의 요약정보를 찾는데도 리용될수 있다. 상위블로크완충기로는 그 어떤 변화가 이루어 질 때만 접근하여야 한다. m_sb_lock에 의하여 변화가 보호되는 핵심안 상위블로크의 마당들은 올려태우기구조체의 잠금을 회전한다. 이 잠금은 탐색되고 있는것이 일치하다는것을 담보하는데 리용될수 있다. 핵심안 상위블로크를 변경시키는 코드는 거래가 상위블로크결속을 한후에 갱신을 보호하기 위하여 이 잠금을 리용한다.

상위블로크관리대면부

xfs_trans_mod_sb() xfs_trans_mod_sb()의 호출은 상위블로크를 즉시 잠금하지 않고 상위블로크의 계수기들을 변경시키는데 리용된다.

함수의 원형은 다음과 같다.

```
Void xfs_trans_mod_sb(xfs_trans_t*tp, unit field, int data);
```

마당의 인수들은 델타파라메터에 넘겨 진 계수기가 더해야 한다는것을 정의한다. 열기를 진행하기 위하여서는 주어 진 계수기로부터 부의 값이 델타에 넘겨 져야 한다.

마당파라메터의 유효값들은 다음과 같다.

▼ XFS_SB_ICOUNT: delta를 sb_icount마당에 준다.
- XFS_SB_IFREE: delta를 sb_ifree마당에 준다.
- XFS_SB_FDBLOCKS: delta를 sb_fdblocks마당에 준다.

▲ XFS_SB_FREXTENTS: delta를 sb_frextents마당에 준다.

xfs_trans_commit()가 호출될 때 상위블로크완충기는 거래에 의하여 잠그어 지며 정의된 델타(delta)모두가 함수에 적용된다. 델타는 루적되며 따라서 같은 마당이 주어 진 거래안의 xfs_trans_mod_sb()에 대한 다중호출로 정의될수도 있다. 일단 거래를 결속하면 델타는 상위블로크의 핵심안 복사에 적용될수도 있다.

xfs_trans_getsb()호출은 거래안에서 상위블로크완충기를 잠금하는데 리용된다. 이 조작은 거래가 xfs_trans_mod_sb()의 호출에 의하여 변경될수 있는것보다는 다른 상위블로크의 마당들을 변경시키는것이 필요할 때만 리용될수 있다. 이 함수의 결과는 xfs_trans_bread()와 꼭 같지만 상위블로크를 얻는데만 리용된다.

함수의 원형은 다음과 같다.

```
buf_t*xfs_trans_getsb(xfs_trans_t*tp);
```

완충기는 xfs_trans_brelse()의 호출에 의하여 개방될수 있다. 특정한 호출이 필요되는 일은 없다.

xfs_getsb()의 호출은 그것이 거래의 앞에서 리용될수 있다는것을 제외하고는 xfs_trans_getsb()와 꼭 같다. 이 함수는 잠금과 상위블로크완충기를 귀환시킨다. 그러나 이 완충기는 절대로 변경시킬수 없다. 왜냐하면 상위블로크는 거래내에서만 갱신될수 있기때문이다. 정보가 요구한 시간의 대부분은 핵심안 상위블로크로부터 리용될수 있으며 따라서 이 함수의 리용은 제한되고 있다.

함수의 원형은 다음과 같다.

```
buf_t*xfs_getsb(xfs_mount_t*mp);
```

xfs_mod_iucore_sb()은 상위블로크의 핵심안 복사를 변경시키는데 리용된다.

이 함수의 원형은 다음과 같다.

```
int xfs_mod_incore_sb(xfs_mount_t*mp,unit field,int delta);
```

마당파라메터는 상위블로크마당이 주어 진 델타에 적용된다는것을 지적한다. 이 부분프로그람은 핵심안 상위블로크를 보호하는 회전잠금을 조심히 다룬다. 현재 이 프로그람은 우에서 정리한 xfs_trans_mod_sb()를 통하여 리용할수 있는 마당의 갱신을 지원하지만 필요하면 그 기능을 확장할수 있다.

이 프로그람은 단위블로크의 계수기들이 절대로 령이하로 될수 없다는것을 강하게 가정하고 있다. 만일 델타가 그런 조건을 발생할수 있게 정의되면 그 델타는 적용될수 없으며 EINVAL을 귀환하게 된다.

xfs_mod_incore_sb_batch()의 호출은 상위블로크의 다중델타를 다중마당에 적용하는데 리용한다. 이 함수는 매개가 마당과 그 마당의 델타를 정의하는 xfs_mod_sb_t구조체의 배렬을 취한다. 다중델타를 정의하기 위하여 호출자가 상위블로크에 그것을 적용할수 있게 함으로써 이 프로그람은 다중델타가 자동적으로 적용될수 있게 하며 xfs_mod_incore_sb()에 대한 호출을 다중화하는데 필요한 잠금휴지시간이 줄어 들게 한다. 함수의 원형과 xfs_mod_sb_t정의는 다음과 같다.

```
typedef struet xfs_mod_sb

unit msb_field ; /* msb_delta를 적용하기 위한 마당*/
int msb_delta ; /*정의된 마당에 더해야 할 량 */
xfs_mod_sb_t
xfs_mod_incore_sb_batch(xfs_mount_t*mp,xfs_mod_sb_t
*msb,uint nmsb);
```

xfs_mod_incore_sb()와 같이 이 부분프로그람은 상위블로크를 보존하는 잠금을 조종하며 상위블로크의 계수기가 령으로 되지 않게 강한 제한을 준다. 만일 정의된 임의의 델타가 그러한 조건을 만족시키면 적용될 델타는 하나도 없으며 함수는 EINVAL을 귀환시킨다.

디스크구조

XFS의 설계에서 공간배치구조의 선택은 대규모파일과 파일체계의 효과적지원요구에 따르는 영향을 받았다. 앞에서 설명한것처럼 공간관리기는 파일안의 블로크배치구조의 최량화, 매 파일의 보증정보의 결정, 디스크상에서 서로 가까운 관계에 있는 파일들의 보존에 관한 역할을 수행한다. 공간관리에 대한 구체적인 내부자료는 사용자가 파일을 충분히 련속적으로 배치하였는지 아니면 보다 련속적으로 배정할 파일공간이 있는지를 결정할수 있게 하는것을 제외하고는 사용자로부터 숨겨 지며 또 이름관리기계층으로부터도 숨겨 진다. 제기된 모든 대면부들은 호출에 기초하며 통보문에는 기초하지 않는다. 조종과 관리통보는 다른 마디로부터 볼수 있으나 체계호출과 관리층에 의하여 조종되게 되며 국부호출로 귀환된다.

디스크상의 구조

상위블로크는 모든 파일체계정보의 뿌리이다. 이 블로크는 파일체계의 앞부분에 배치된다(변위 0, Linux하에서는 LILO와 충돌할수 있기때문이다.). 이것은 전통적인 UNIX 파일체계의 속성과는 차이난다. UNIX는 변위 512에서 출발한다.

XFS와 다른 Linux파일체계와의 혼돈을 피하기 위하여 XFS파일체계의 변위 512는 다른 파일체계의 상위블로크의 매지크번호와 다른 값을 포함해야 한다. 상위블로크는 파일체계의 다른 모든 요소들을 찾는데 충분한 정보를 가지고 있다. 올려태우기된 파일체계에 속하는 정보부분인 상위블로크의 한개의 복사판이 있다.

다음의 마당들은 상위블로크에서 가장 중요한 마당들이다.

▼ XFS매지크번호
- XFS판본
- 파일체계의 유일id

- 마지막으로 올려태우기된 파일체계이름
- 론리적블로크크기(lbsz바이트크기로 29..216)
- 신뢰할수 없는 범위크기(lbsz로)
- 물리적쩩터크기(바이트)
- 색인마디크기(바이트, 27..211)와 색인마디의 매 자료대역과 속성대역의 최소크기와 같이 색인마디의 공간을 어떻게 분할하는가에 대한 정보
- 자료블로크배정기술, 비트매프의 선택이나 B나무, 기타
- 색인마디에 배정된 작은 파일
- 배정그룹크기(lbsz)
- 총 파일체계자료부분－기록권크기(lbsz로)
- 총 파일체계의 신뢰할수 없는 부분기록권크기(extent단위로)
- 신뢰할수 없는 부분기록권범위에 대한 비트매프의 론리적블로크번호
- 신뢰할수 없는 부분기록권비트의 요약정보론리적블로크번호
- 배정 혹은 자유색인마디의 총수
- 빈 자료부분 기록권블로크의 총수

▲ 자료와 자료부분기록권의 메타자료로 배정된 블로크들의 총수

표준조작시에 변화되는 마당들은 오직 df에 의하여 리용된 정보를 포함하는 정적마당들만이다.

이 정보들의 변화는 재기동과정에 정보를 받을수 있는지 없는지에 관한 파일체계의 일관성보존을 위하여 가동일지에 기록되여야 한다. 바꾸어 말하면 올려태우기시에 정보가 계산될수 있으나 디스크에는 전혀 기록될수 없다(올려태우기시를 제외하고). 이것은 올려태우기시에 파일체계의 모든 배정구조체를 주사하는 비용으로 상위블로크의 변화를 가동일지에 기록하는 휴지시간을 피하자는데 있다. 현재 계획은 올려태우기의 주사를 피하고 상위블로크의 변화를 기록하자는데 있다. 파일체계크기와 전체 마당도 역시 파일체계의 크기가 재정의될 때 동적으로 변화된다. 이 변화들은 기록되여야 한다.

배정그룹머리부

매개 파일체계자료부분－기록권은 같은 크기의(제일 마지막의것을 제외) 배정그룹들로 분할된다. 이 크기는 파일체계가 생성될 때 선택된다. 크기는 전체 파일체계크기를 8개로 나눈 기정크기로서 16MB~1GB범위에 있게 된다. 중요한 최소배정그룹크기의 조건으로서 8개의 배정그룹들중에는 최소값이 존재하게 된다. 또한 배정그룹크기들에 대한 배정과정에 둥그리기도 있을수 있다. 이것에 대한 구체적인 내용은 계속 연구하고 있다.

파일체계를 배정그룹으로 나누는 첫번째 리유는 파일체계의 공간배정에서 병렬화를 촉진시키자는것이다. 배정정보에 관한 잠금조작은 매 배정그룹당 따로따로 진행할수 있으며 이것으로 하여 성능이 개선되는데 특히 다중처리기환경에서 성능이 개선된다. 배정

그룹들은 읽기불가능한 환경에서도 그것들을 보다 쉽게 찾을수 있는 유일한 크기를 가진다. 만일 배정그룹들이 가변크기를 가진다면 매개 블로크는 다음 블로크가 어디에 있는가를 계산해 낼수 있도록 읽기가능한 상태에 있든지 혹은 모든 배정그룹들의 주소를 포함하는 첨수여야 한다. 매개 배정그룹은 다음과 같은것들로 구성된다. 0바이트위치의 상위블로크(첫번째 블로크는 파일체계가 생성된후에 갱신된 값을 취한다.), 배정그룹머리부(상위블로크자료에 따라 첫번째 블로크에 일치시킨), 배정그룹머리부에 의하여 지적되는 자료, 파일체계메타자료에서 모든 《지적자》들은 배정그룹의 시작과 관계되는 32bit블로크번호 아래의것을 제외하고 64bit론리적블로크번호이다. 다음 마당들은 배정그룹머리부에 주어 진다.

▼ 배정그룹머리부 매지크번호(검사용)
- 배정그룹머리부 판본번호
- 령으로부터 출발하는 배정그룹번호
- 비트매프배정도식이 리용될 때 자유블로크비트매프와 요약정보의 위치와 관계되는 블로크번호와 크기(lbsz)
- B나무배정도식이 리용될 때 B나무 매 뿌리의 위치(관계되는 블로크번호) 배정그룹머리부의 론리적블로크로 한개 혹은 두개의 뿌리를 일치시키기 위하여 선택할수도 있다.

▲ 색인마디표를 포함하는 《색인마디》의 위치(관계되는 블로크번호)대신 배정그룹의 다음것을 기억시킬수도 있다. 나무와 배정된 블로크와 색인마디들은 매 배정그룹별로 유지될수 있다.

정확성을 위하여 배정그룹머리부를 가동일지에 기록해야 할 필요가 없는 여유가동일지등록을 포함하여 변경사항들이 기록되여야 한다.

한편 이 정보는 오직 상위블로크에만 존재하게 되며 df의 목적을 달성하는데는 충분하다.

자료블로크자유목록

자료블로크배정을 고찰하는데는 대체로 두가지 도식이 있다. 두 경우에 설계는 핵심부대역 즉 완충기에 아무러한 정보도 보존하지 않는다. 즉 모든 정보는 디스크로부터 읽어 들이고 또 디스크에 쓰며 모든 변경내용은 가동일지에 기록된다. 이 사실은 매 배정그룹별로 기억을 소비하는 설계보다도 기억리용에 관해서는 더 유연성 있는 설계로 되게 한다.

첫 도식에서 비트매프는 머리부정보와 비트매프자체를 포함한 배정그룹안의 모든 론리적블로크들을 포괄한다. 비트매프는 배정그룹블로크로부터 얻어 진 론리적블로크들의 단순한 범위이다. 비트매프는 파일체계가 확장(파일체계의 가장 오랜 마지막 배정그룹)되거나 축소(파일체계의 제일 최근의 배정그룹)될 때만 옮겨 진다. 비트매프에서 비트들은 자유블로크들의 모임이며 배정블로크에 관해서는 명백하다. 주어진 크기의 빈 범위를 찾기 위하여, 전체 비트매프를 주사하는 현상을 피하기 위하여 보충적인 정보가 기억된

다. 매개 비트매프블로크에서 2의 루승(2KB)인 배정그룹에서의 매개 가능한 범위에 관하여 그 블로크로부터 시작하여 2KB~2KB+1−1의 크기의 빈 범위수를 보관한다. 《블로크》는 파일체계의 론리적블로크들이다. 이 정보는 파일체계의 배정그룹과 론리적블로크의 크기에 따라 가변크기의 공간을 차지한다. 512byte블로크들과 1GB의 배정그룹에 관한 최악의 경우는 21KB이다. 4KB와 1GB의 배정그룹에 관하여 정보의 크기는 288byte이다(4KB는 비트매프블로크크기로 주어 진다고 가정).

매개 계수기입구점은 16bit이다. 이 정보는 주어 진 크기에 관계된 모든 입구점들이 함께 있을수 있게 순서화된다. 이 입구점들은 요청을 만족시킬수 있게 충분하게 큰 빈 범위를 서술해야 하는 비트매프블로크를 탐색한다. 다음 비트매프블로크는 출발위치를 찾기 위하여 탐색한다. 이 도식을 약간 개조하여 매개 계수값이 단일한 비트매프블로크안의 빈부분만을 포함하도록 제한할수 있다. 이 도식은 성능에 의존하여 선택할수 있다.

두번째 도식에서 배정정보는 B나무의 쌍에 보존된다. 두개의 B나무는 배정그룹안의 모든 빈 범위에 대하여 그 쌍들을(시작자유블로크, 자유블로크계수값)자료로서 포함한다. 한개 B나무는 시작자유블로크로 첨수화되며 다른 나무는 자유블로크계수값에 의하여 첨수화된다. 2차적으로 열쇠를 일의적으로 생성하기 위하여 빈시작블로크에 의하여 첨수화한다. 블로크배정은 처음에 한개 B나무를 탐색하며 다음 캐쉬의 두 B나무를 갱신하며 가동일지를 변경시킨다. 일단 가동일지입구점이 만들어 지면 B나무완충기들은 실제적으로 디스크에 쓰기 위하여 개방된다. 캐쉬가 기본적으로 매타자료용의 LRU도식을 실현하고 있다고 가정하면 이것은 실제적으로 참조되고 있는 블로크들만이 기억에 존재한다는것을 의미한다.

색인마디표

XFS설계자들은 전적으로 색인마디를 배정하는 전통적인 방법(solaris의 UNIX파일체계인 UFS와 같은)이 XFS의 기본적인 설계목적과 배치된다는것을 알았다. 따라서 XFS는 색인마디를 요구대로 작은 그룹으로 배정하는 도식을 리용한다. 이 방법은 색인마디들이 단일한 가변크기범위로 기억되거나 혹은 색인마디의 묶음을 지적하는 고수준첨수라는것을 말해 준다. 단일범위도식은 단순하지만 파일체계가 토막화되면 배정이 실패할 우려가 있으므로 리용하지 않는다.

첨수도식은 두가지 일반적인 방법들중에서 어느 하나로 동작할수 있다. 즉 고정크기묶음의 자유색인마디와 단일범위첨수방법이든가 혹은 정규파일에서 리용되는것처럼 색인마디《파일》용의 B나무(혹은 순서화된 범위지적자)리용방법으로 동작할수 있다. 여기서는 후자의 방법을 선택하였다. 매 배정그룹의 머리부는 색인마디공간과 색인마디번호를 표시하는 B나무의 뿌리를 포함한다. B나무는 배정그룹을 위한 모든 색인마디공간을 포함하는 《파일》을 표현하며 색인마디번호는 배정그룹의 색인마디번호목록에서 첫번째 색인마디를 가리킨다. 색인마디자유목록상의 색인마디들은 색인마디마당을 통하여 색인마디번호로 련결된다. 색인마디들을 포함하는 《파일》은 필요할 때 가능하면 선행배정에 대하여 확장된다. 임의의 다른 파일에서 진행되는바와 같이 색인마디들을 련속적으로

확장하려는 시도들이 있다. 하지만 색인마디에 대한 직접 접근은 드물며 따라서 그의 실현이 그리 중요하지 않다는데 대하여 강조한다.

색인마디표의 범위수를 작게 보존하는 실제적리유는 그것을 보다 넓게 표현할수 있는 B나무를 보존하는데 있다. B나무입구점은 색인마디범위의 첫번째 색인마디에 대한 색인마디번호의 낮은 32bit와 범위안에서의 색인마디번호(항상 한 블로크안의 색인마디의 다중화된 번호), 관계되는(배정그룹머리부와) 범위시작의 디스크블로크번호를 포함한다. B나무는 색인마디번호마당에 의하여 첨수화된다. 이것은 마당의 크기와 단위로만 bmap함수에 리용된 B나무와는 다르다. 그러나 알고리듬은 같다. 자유목록리용에 비한 색인마디비트매프의 리용상 문제점은 실제로 색인마디관리도식의 비교에서 서로 독립적인 차원에 있다. 비트매프의 경우에 배정과 해제는 둘다 색인마디와 비트매프단어를 변경시킬것을 요구한다. 자유목록의 경우에 배정과 해제는 둘다 배정그룹머리부와 색인마디의 변경을 요구한다. 자유목록의 경우는 자유목록의 지적자가 색인마디에 기억되기때문에 전체적으로는 더 적은 기억을 요구한다. 리론적으로 자유목록의 경우는 비트매프가 서로 독립적으로 잠금이 수행될수 있는 쪼각들로 나누어 질수 있기때문에 비트매프의 경우보다 더 적은 병렬처리를 요구한다. 사실상 이것은 그리 크지 않은 제한성이다. 왜냐하면 색인마디배정은 매개 배정그룹에서 병렬적으로 진행되기때문이다. 비트매프의 경우는 배정그룹들이 서로 더 쉽게 가까와 지게 한다. 그러나 이것이 어느 정도 중요한가 하는것은 명백하지 않다.

자유목록대신에 비트매프도식을 사용함으로써 대다수의 설계들은 색인마디《파일》에 걸쳐 고유한 간격으로 자유색인마디비트매프의 색인마디크기화된 묶음들을 확산시킨다. 실례로 색인마디크기가 256byte이면 256×8=2048의 색인마디간격을 가지게 되며 색인마디《파일》의 변위자료는 색인마디대신 비트매프의 요소로 될것이다. 따라서 배정된 색인마디에 《가까운》 자유색인마디를 찾고 계산하기 위하여 적당한 비트매프블로크를 읽고 배정된 색인마디비트위치의 근방을 찾아야 한다.

다른 가능한 설계는 비트매프에 대한 범위의 독립적인 모임을 가지는것이다. 그러면 우연적인 크기를 가지는 비트매프들의 취급에 대하여 총체적으로 론의하여야 한다. 비트매프를 작은 범위 즉 고정하거나 가변적인 크기로 전환하는것이 실천적이다. 어떤 경우에 지적한 비트매프블로크를 찾는 일을 무시할수 없다고 하면 보다 많은 I/O가 비트매프를 조종하는데 요구되며 자유목록에 비한 단순한 비트매프의 부족점은 대부분의 색인마디들이 배정되였을 때 정상상태에서 자유색인마디를 찾는 일이 보다 더 어렵다는데 있다.

색인마디번호

색인마디는 파일체계안의 매개 파일, 등록부 등을 정의하는 정보들을 가지고 있다. 매 색인마디는 색인마디번호에 의하여 혹은 첨수번호(inumber)에 의하여 이름 지어 진다. 전통적인 UNIX파일체계들에서 색인마디는 전체 파일체계에 걸쳐 순차적으로 번호화되여 있다. 그러한 체계들에는 매개의 배정 혹은 실린더그룹에 같은 수의 색인마디가 있으며

따라서 특정한 색인마디를 찾는데서는 지장이 없다. XFS의 매개 배정그룹에는 가변길이의 색인마디들이 있으며 따라서 전통적인 번호화도식은 첨수번호를 색인마디의 디스크주소로 변환하기가 대단히 힘들다는것을 말해 준다. XFS에서 색인마디번호는 두개의 비트마당으로 분할된다. 보다 중요한 비트마당은 배정그룹번호이고 덜 중요한 비트마당은 배정그룹안의 색인마디번호이다. 현재까지는 두개의 비트마당이 각각 32bit이고 첨수번호는 64bit정수이다. 32bit정수를 배정그룹번호와 색인마디번호로 나누는데서 어려운 문제는 파일체계가 너무 커질수 있는것이며 그것으로 인하여 일부 체계는 비트마당의 공간밖에서 실행되게 된다. 현재는 이 파일체계의 설계가 10년가량은 그대로 리용되리라고 보고 디스크양식에서 이 공간을 쓰기로 하였다.

자료와 속성블로크표현

기록권안에서 파일의 주소공간을 디스크블로크로 넘기는것은 B나무나 혹은 범위서술자에 의하여 실현된다. B나무에서 나무의 매개 마디의 정보는 파일시작변위(론리적블로크에서), 기록권블로크시작번호, 범위길이(론리적블로크)이다.

B나무는 다중준위로 구성된다. 뿌리준위를 제외하고 매 준위에는 다중블로크들이 있다. 잎이 없는 매개 블로크는 열쇠(파일의 시작변위값)와 다른 블로크에 대한 지적자를 포함한다.

B나무는 파일시작변위마당우에서 열쇠화된다. B나무의 뿌리블로크는 색인마디에 기억되는데 이것은 뿌리블로크가 B나무의 다른 블로크와 크기가 다르다는것을 의미하며 파일체계의 론리적블로크라는것을 의미한다. 지적자가 두개의 열쇠사이에 놓일 때 지적자에 의하여 지적된 블로크의 자료(파일의 시작변위)는 두 열쇠값사이에 놓인다. 매개 블로크는 K개의 열쇠와 K+1개의 지적자를 가지고 있다.

블로크가 완전히 차고 삽입이 요구될 때 두 조작들중 하나의 조작이 수행된다. 첫째로 회전이 시도되는데 회전은 린접하는 두 블로크들사이로 파잉마디를 이동시키려고 한다. 만일 블로크들이 차 있으면 블로크는 분할되고 정보의 절반은 새 블로크에 옮겨 지며 상위블로크는 한개대신 두개의 블로크를 지적한다. 다른한편 두개 블로크들은 세개의 새로운 블로크들을 생성하면서 분할할수 있다. 이와 같은 방법으로 나무가 계속 무성해진다. 수속은 뿌리에 도달할 때까지 혹은 상위블로크에 새로운 정보를 위한 자리가 마련될 때까지 재귀적으로 련속된다. 지우기조작은 본질적으로 남아 있는 블로크들이 충분히 차 있는 정도에 따라 반대로 진행한다. 탐색조작은 나무에서 같은 블로크번호라든가 혹은 가장 가까우면서 보다 낮은 객체를 찾으며 요청된 다음블로크가 존재하는가를 검사한다.

범위서술자의 경우에 우리는 배렬의 쌍 즉 범위크기배렬(lbsz, 32bit)과 범위에 대한 지적자배렬(64bit블로크번호)을 가지고 있다. 만일 파일에 구멍이 있으면 령범위지적자에 의하여 표시된다. 이 도식은 작은 파일범위에 의하여 전체 파일에 리용될수 있다. 도식은 파일변위정보가 루적크기에 의하여 암시적으로 표시되기때문에 특정한 블로크를 찾자면 배렬전체에 대하여 선행탐색을 요구하게 된다.

다른 방법으로서 파일변위는(다른것은 64bit) 기억되기도 하며 파일의 구멍에 대하여 령지적자를 무시할수도 있다. 이것은 2진탐색이 가능하다는것을 의미한다. 다른 말로 말하여 파일들이 구멍을 가지지 않는 표준경우에는 더 적은 사용자가 적합할수 있다는것을 의미한다. 이 정보는 색인마디등록부에 기억된다. 이 정보를 128bit아래로 축소시키기 위하여 정보를 기억시킬수 있다. 실례로 범위에 해당하여 21bit, 기록권블로크번호에 해당하여 52bit, 파일블로크번호에 해당하여 55bit로 기억시킬수 있다.

범위서술자조작은 그 표현이 색인마디에 일치하는 조건에서는 사용가능하다. 사용할수 없는 부분—기록권은 여러개의 고정크기쪼각으로 나누어 진다. 그 크기는 파일체계다중블로크크기이며 mkfs시에 설정되고 상위블로크애 기억되며 1MB정도 상대적으로 크다. 따라서 이 부분—기록권에서 빈 공간은 단순한 비트매프에 의하여 표시된다. 비트매프는 신뢰성이 있어야 하며 그 내용은 자료부분—기록권에 기억되는데 상위블로크가 그것을 지적한다. 배정속도를 더 높이기 위하여 계수기에는 매 비트모임의 비트매프블로크번호별로 보존된다. 이 계수값모임은 자료부분—기록권에 범위로써 보관되며 또한 상위블로크에 의하여 지적된다.

파일체계의 구조

매개 올려태우기된 파일체계에는 파일체계에 부속되는 정보를 포함하거나 지적하는 핵심안 구조체(배정된)가 있다(VFS구조체). 이 구조체는 파일체계실현을 위한 전용자료인 한개의 마당 cfs_data를 포함하고 있다. 이 마당은 몇가지 파일별 정보를 포함하는 한개의 구조체(XFS용의 xfs_mount)를 지적한다.

xfs_mount구조체는 다음의 정보를 포함한다.

▼ VFS구조체에 대한 거꿀지적자
- 기록권의 자료대역에 대한 블로크장치용Vnode지적자
- 기록권의 가동일지지역에 관한 블로크장치용Vvnode지적자
- 파일체계의 핵심안 뿌리색인마디에 대한 지적자
- 배정몫에 대한 몇가지 마당 ; EFS에는 기발, 색인마디지적자, 크기가 있다.
- 체계실현을 전환하여 자체사용에 쓰기 위한 몇가지 통계적값
- 상위블로크구조체의 복사

▲ 배정그룹당 한개씩의 짧은 구조체배렬

배정대완충화

림계자료구조체는 핵심안 배정대완충화에 대한 몇가지 질문을 가지고 있는데 여기에는 색인마디, 자료, 색인마디배정비트매프들이 포함된다. 색인마디에 대하여 색인마디를 핵심안이나 핵심안 색인마디구조체의 고정크기풀에 캐쉬에 기억하는 우선권순위가 반드시 존재한다. 핵심안 색인마디는 디스크상의 색인마디뿐아니라 지적자와 다른 정보도 포함한다. 이 색인마디풀을 위한 배정방법은 검증되여야 한다. 색인마디에 관하여 다음의

의문은 파일의 디스크상의 구조를 표현하는 B나무가 기억에 넣어 지는가 혹은 요구상으로 완충기에 기억되는가 하는것이다. 대규모파일체계를 지원하기 위하여 이 방법은 완충기를 리용할것을 요구한다. 비트매프를 참조하는데 완충기를 리용하도록 지시하는 빈자료블로크비트매프(혹은 B나무)는 잠재적으로 매우 크다. 대규모파일체계용으로 배정된 기억에 비트매프를 복사하는 기능을 사용자가 제공할수 있다고 볼수 없다. 색인마디배정구조체는 크지 않지만 성능상 본질적손실이 없이 완충화될수 있다. 이러한 가정은 실천적으로 검증되여야 한다.

XFS의 유용성과 새 판본예고

SGIXFS pre-release 0.9는 Linux 2.4.0에 대하여 검사수정으로서 리용할수 있다. 또한 이것은 RPM의 모임으로서 그리고 Modified Red Hat 설치자로서 리용할수도 있다. 다른 구획에서 Modified Red Hat 7.0설치자는 뿌리구획이나 혹은 다른 구획에서 XFS로 Red Hat 7.0체계를 설치하기 위한 RedHat 7.0설치매체를 가지고 작업한다.

Linux용XFS파일체계를 설치하기전에 제한목록, 요구 그리고 이 발표판에 해당한 특정한 지령들에 관하여 Linux pre-release 0.9의 XFS를 일일이 알아 보아야 한다.

XFS에 의한 작업

Linux 워크스테이션이나 더 좋기는 봉사기상에서 XFS로 작업하기 위하여서는 이를 위한 구획과 파일체계를 다시 생성하여야 한다. ext2파일체계로부터 직접적인 이행은 허용되지 않는다. XFS로 작업하기 위하여서는 3가지 단계 ; 구획설정, 양식화, 올려태우기가 필요하다.

구획설정

새로운 XFS파일체계를 생성하기 위하여서는 구획이 필요하다. 이 구획은 새로운 디스크로부터 만들수도 있고 이미 존재하는 디스크상의 설정되지 않은 구획공간으로부터도 만들수 있으며 현존구획에 중복쓰기하여 설정할수도 있다. 일반적으로 생성하기 위하여서는 fdisk명령을 사용하여 "Linux Native(83)"으로 구획을 설정할수도 있고 그 구획우에서 XFS파일체계를 만들기 위하여 다음의 명령들을 리용할수도 있다.

XFS파일체계생성

어떤 다른 Linux파일체계를 다음의 명령을 리용하여 생성했던것처럼 같은 방식으로 새로운 XFS파일체계를 생성할수 있다.

```
mkfs -t xfs/dev/<devfile>
```

여기서 /dev/<devfile>은 파일체계를 생성하려고 하는 구획이다. 이 과정이 구획에 현재 존재하는 임의의 파일체계를 파괴한다는데 대하여 강조한다.

실례로 2차 SCSI구동기의 세번째 구획에 파일체계를 생성하기 위하여 다음의 명령을 사용할수 있다.

```
mkfs -t xfs/dev/sdb3
```

요구될수 있는 중요한 한가지 선택항목은 "−f"인데 이 항목은 현재 구획에 이미 파일체계가 존재할 때 새로운 파일체계의 생성을 강하게 요구하게 된다. 역시 여기서도 구획상에 현재 있는 모든 자료는 파괴될것이라는것을 강조한다.

```
mkfs -t xfs-f/dev/<devfile>
```

보다 더 좋은 성능을 얻기 위하여서는 가동일지파일의 크기를 기정값 1, 200블로크로부터 8, 000블로크까지 증가시켜야 한다.

다음의 명령은 파일체계를 생성하는 방법으로 실현할수 있다.

```
mkfs -t xfs-1 iuternal, size=8000b −d name=/dev/c<devfile>
```

다른 선택항목들은 XFS파일체계생성에 리용할수 있다.

XFS파일체계의 올려태우기

다음으로 새로운 파일체계를 올려태우기해 보겠다.

올려태우기명령은 다음과 같다.

```
mount −t xfs/dev/<devfile>/<mount_pt>
```

여기서 /dev/<devfile>은 파일체계를 포함하는 장치이며 /<mount_pt>는 파일체계를 위한 올려태우기점이다. XFS는 실행기록파일체계이므로 파일체계를 올려태우기하기전에 완료되지 않은 임의의 거래에 관한 거래가동일지를 검사하며 새로운 파일체계로 이동시켜 준다.

부록 1. 쏘프트웨어 RAID의 리용방법

이 리용방법은 Linux상태에서 쏘프트웨어 RAID를 어떻게 사용하는가에 대하여 서술한다. 이 체계는 쏘프트웨어 RAID층의 특정한 판본 즉 Ingo Molnar나 다른 업체들에서 제작한 0.90RAID층 등을 식별할수 있다. 여기서 설명하는 체계는 Linux2.4에서 표준으로 되며 Linux2.2핵심부에서 리용할수 있는 판본이다. 0.90RAID지원판은 Linux 2.0과 Linux2.4에 검사수정으로써 리용할수 있으며 여러가지 고찰결과에 의하면 이미 이 판본들에서 리용하는 변경된 RAID지원판보다 훨씬 더 안정하다는것이 인정되였다.

1. 요 약

2.0과 2.1핵심부의 표준으로 되고 있는것들중의 하나인 변경된 raid층에 대하여 서술하기 위하여서는 Linuxdoc. org의 Linux Docameutation Project로부터 리용할수 있는 Linas Vepstas(linas@linas.org)에서 HOWTO를 찾아 보면 된다.

이 참고서를 위한 홈싸이트는 http : //ostenfeld.dk/~jakob/software−RAID. HOWTO/이며 여기에는 우선 갱신된 판본들이 공개되여 있다.

이 참고서는 RAID개발자들과 여러 연구자들의 의견교환에 기초하여 씌여 졌는데 그것을 쓰게 된 리유는 쏘프트웨어 RAID 참고서가 이미 존재한다고 해도 변경된 참고서는 표준 2.0, 2.4핵심부에서 볼수 있는 변경된 형식의 쏘프트웨어 RAID를 서술하고 있기때문이다. 이 참고서는 보다 최근에 개발된 새 형식인 RAID의 리용에 대하여 서술한다. 새 형식의 RAID는 변경된 형식의 RAID에는 없는 수많은 특성을 가지고 있다.

우리가 2.0이나 2.4핵심부로 새 형식의 RAID를 리용하려고 하면 핵심부를 위한 검사수정을 얻어야 하는데 ftp://ftp.[나라코드].Kernel.org/pub/linux/daemons/raid/alpha이라든가 혹은 http://people.redhat.com/mingo/로부터 보다 최신자료를 얻을수 있다. 표준 2.2핵심부는 이 참고서에서 서술한 새 형식의 RAID를 직접 지원하지 못한다. 따라서 이러한 검사수정이 요구된다. 표준 2.0과 2.2핵심부의 변경된 형식인 RAID는 오유가 있고 새 형식의 RAID프로그람에 주어 진 일부 중요한 특성들이 결여되였다.

여기서 서술하는것처럼 새 형식의 RAID지원은 2.3개발판핵심부에 보충되고 있으며 따라서 2.4Linux핵심부에로 주어 지게 될것이다. 그러나 완성되기까지는 안정판핵심부를 수동으로 검사수정하여야 한다.

2.2에서 RAID를 지원하기 위하여 알란 콕스(Alan Cox)가 발표한 −ac핵심부를 사용하려고 해도 좋다. 이것들중의 많은것은 새 형식의 RAID에 포함되여 있으며 핵심부를 자체로 검사수정하는 작업으로부터 사용자의 안전을 보장할수 있다.

이 참고서에 대한 정보들중에서 그 일부는 대수롭지 않게 보이지만 raid를 다 알고 있으면 충분하다.

1.1. 권 고

RAID가 많은 사람들에게 안정하게 보이더라도 유익하게 동작하지 못할수도 있다. 만일 자료와 공정을 모두 잃어 버렸다면 마치 자동차충돌사고와 같이 우연적사고로 하여 어쩔수 없는것처럼 우리와 개발자들의 잘못은 아니라고 보아야 할것이다. 그리하여 RAID쏘프트웨어와 정보에 대하여서는 사용자가 책임져야 한다. 이런 점에서는 쏘프트웨어나 정보도 모든 면에서 정확한것이라고 담보할수 없으며 어디에나 다 리용할수는 없다. 때문에 이에 대하여서는 실험을 거쳐야 보다 안전하게 쓸수 있다. RAID쏘프트웨어와 관련된 안정성문제가 아직까지는 제기되지 않았다는것을 강조해 둔다.

1.2. 요 구

이 참고서는 사용자들이 raid0145를 정합시키는 검사수정과 raid도구의 0.90판본을 가진 2.2x 혹은 2.0x핵심부의 신판을 사용하거나 2.3핵심부의 후속판(판본 >2.3.46)을 사용하다가 마침내는 2.4판을 사용하게 된다.

검사수정과 도구는 다같이 ftp://ftp.fi.kernel.org/pub/linux/daemons/raid/alpha 그리고 일부 경우에는 http://peopel. redhat.com/mingo/에서 찾아 볼수 있다.

RAID검사수정, raid도구모임 그리고 핵심부는 가능한대로 모두 밀접하게 일치시켜야 한다.

간혹 raid검사수정들을 맨 마지막핵심부에 리용할수 없다면 변경된 핵심부를 사용해도 된다.

2. RAID를 쓰는 리유

여러가지 리유로 하여 RAID를 리용할수 있다. 그러한 리유로는 물리적디스크를 하나의 보다 큰 《가상》장치로 결합시키는 능력과 성능의 향상, 여유도 등을 들수 있다.

2.1. 전문절차

Linux RAID는 대부분의 블로크장치들에서 작업할수 있다. 사용자가 IDE를 쓰든지 SCSI장치를 쓰든지 또 그것들을 섞어 써도 아무런 일이 없다. 일부 사람들은 다소간의 성공률을 가지고 망블로크장치(Network Block Device : NBD)를 리용하려고도 한다.

장치에서의 모선들도 속도가 충분히 빠르다는것을 담보할수 있다. 사용자는 매개 장치가 10MB/s를 제공할수 있고 모선이 40MB/s만 유지할수 있을 때 한대의 UW모선에 14개의 UW9-SCSI구동기를 련결할수 없다. 또한 사용자는 IDE모선당 한개의 장치만을 리용할수 있다. 디스크들을 주종방식으로 실행시키는것은 사실상 성능상 견지에서 보면 그닥 좋은 방법이 못된다. IDE는 모선당 한개이상의 장치에 대한 접근에서는 실제적으로 불리하다. 물론 보다 새로운 주기판들은 모두 두개의 IDE모선을 가지고 있으며 따라서 조종장치를 더 주입하지 않고도 RAID에 두개의 디스크를 설치할수 있다.

RAID층은 절대적으로 파일체계층과 해야 될 일이 아무것도 없다. 다른 블로크장치와 같이 RAID우에 임의의 파일체계를 설치할수 있다.

2.2. 용어

용어 "RAID"는 《Linux프로그람 RAID》를 의미한다. 이 참고서는 장치 RAID에 대하여서는 아무런 측면도 취급하지 않는다.

설치할 때는 디스크의 수와 그것들의 치수를 고려해야 한다. 문자 N은 항상 배렬안에서 능동디스크의 수를 표시하는데 리용된다. 문자 S는 특별히 지적하지 않는 한 배렬의 제일 작은 구동기의 크기를 의미한다. 문자 P는 MB/S로 표시되는 배렬안에서 한개 디스크의 성능으로서 리용된다. 리용할 때 우리는 사실과는 맞지 않지만 디스크들의 속도는 항상 같다고 본다.

용어 《장치》와 《디스크》는 같은 객체를 의미하려고 가정된것이라는것을 강조해둔다. 보통 RAID장치를 구성하는데 리용되는 장치들은 디스크상의 구획이며 전체 디스크는 아니다. 하지만 한개 디스크상에 여러개의 구획을 결합하는것은 보통 리해되지 않으며 따라서 장치라는 말과 디스크라는 말은 곧 《서로 다른 디스크상의 구획》을 의미한다.

2.3. RAID의 준위

여기에서는 Linux RAID검사수정에 지원되는것이 무엇인가에 대하여 간단히 서술한다. 이 정보들은 절대적이고 기본적인 RAID정보이지만 Linux실현준위에서 특별한것이 무엇인가에 대한 몇가지 견해를 보충한다. 사용자가 RAID에 대하여 알고 있다면 이 부분은 생략한다. 이제 이미 론의하던 문제에 대하여 상기하자.

현재 Linux용RAID검사수정들은 다음과 같은 준위들을 지원한다.

▼ 선형방식

- 두개이상의 디스크들은 한개의 물리적장치와 결합된다. 따라서 디스크들은 서로 《첨가》하면서 RAID장치에 대한 쓰기는 디스크 0을 채우고 다음에 디스크 1을 채우는 방법으로 차례차례 진행된다. 디스크들이 같은 크기를 가지지 않기때문에 크기는 전혀 문제로 되지 않는다.
- 이 준위에는 여유도가 없다. 한개 디스크가 손상되면 사용자는 자기 자료의 거의 대다수를 잃어 버리게 될것이다. 하지만 사용자는 일부 자료를 회복시킬수도 있다. 왜냐하면 파일체계가 하나의 큰 련속적인 자료덩어리를 놓칠수도 있기때문이다.
- 읽기와 쓰기성능은 단일읽기/쓰기에서 증가하지 않는다. 그러나 여러명의 사용자가 장치를 리용할 때 한 사용자는 첫 디스크를 효과적으로 사용할수 있지만 다른 사용자는 두번째 디스크에 존재할수 있게 어떤 일을 발생시킨 파일로 접근할수 있다. 만일 무슨 일이 생기면 성능을 다시 알아 보아야 한다.

- RAID-0

- 《줄무늬》방식이라고도 부른다. 읽기와 쓰기가 장치들에 대하여 병렬로 진행되는것을 제외하고는 선형방식과 같다. 장치들은 대략적으로 같은 크기를 가져야 하며 모든 접근이 병렬로 진행되기때문에 동일하게 채워 진다. 만일 한개 장치가 다른 장치보다 훨씬 크면 여유공간이 여전히 RAID장치에서 리용되지만 RAID장

치의 고속말단에 쓰기를 진행하는 동안에는 큰 디스크하나에만 접근하게 될것이다. 이 과정은 성능을 떨어 뜨린다.

- 선형방식에서와 같이 이 준위에도 역시 여유도가 없다. 선형방식과는 달리 장치가 고장나면 자료를 구원할수 없다. 만약 설정된 RAID−0으로부터 장치를 제거하면 RAID장치는 한개의 련속적인 자료블로크를 읽지 않게 되며 장치전반에 작은 구멍들을 채워 넣게 된다. e2fsck는 아마 그러한 장치들로부터 많은것을 회복하지 못하게 될것이다.
- 장치상에서 읽기와 쓰기가 병렬로 진행되기때문에 읽기, 쓰기성능이 높아 지므로 RAID−0을 사용한다. 디스크에 대한 처리가 충분히 빠르면 N×P MB/S에 근사한 높은 성능을 얻을수 있다.

- RAID−1

- 여유도를 가지고 있는 첫번째 방식이다. RAID−1은 0 혹은 그이상의 예비디스크를 가지고 있는 두개이상의 디스크상에서 리용할수 있다. 이 방식은 다른 디스크상에 한개 디스크에 해당한 정확한 거울정보를 유지한다. 물론 디스크들은 크기가 같아야 한다. 만일 한개의 디스크가 다른것보다 더 크면 RAID장치는 제일 작은 크기를 가지게 될것이다.
- N–1개까지의 디스크가 제거될 때(혹은 파손될 때) 모든 자료는 그대로 보존되며 예비디스크를 리용할수 있으면 그리고 체계가(실례로 SCSI구동기 혹은 IDE소편등) 폭주상태에서도 살아 있으면 구동기오유검출후 거울의 재구성을 예비디스크상에서 즉시 시작할수 있다.
- 쓰기성능은 자료의 동일한 쓰기가 배렬안의 모든 디스크에 대하여 전송되여야하기때문에 단일장치에서보다 좀 낮을수 있다. 읽기성능은 보통 RAID의 코드에 지나치게 간소화된 읽기균형방법을 적용하는것으로 하여 단일장치에서보다 더 낮다. 그러나 여러가지 보다 갱신된 읽기균형방법들이 실현되였으며 따라서 이 방법들을 Linux2.2 RAID검사수정에 리용할수 있으며 표준 2.4핵심부 RAID 지원기능과 거의 같다.

- RAID−4

이 RAID준위는 그리 자주 리용되지는 않는다. 이 준위에서는 3개 혹은 4개의 디스크를 리용할수 있다. 정보를 완전히 거울화하지 않고 한개 디스크에 기우성정보를 보존하며 RAID−0에서와 같은 방법으로 다른 디스크에 자료를 기억한다. 기우성정보를 기록하기 위하여 디스크들이 예약되기때문에 배렬의 크기는 (N–1)×S로 된다. 여기서 S는 배렬안의 제일 작은 디스크의 크기이다. RAID−1에서와 같이 디스크들은 같은 크기를 가져야 하거나 혹은 (N–1)×S식의 S가 배렬의 가장 작은 구동기의 크기로 되여야 한다.

- 만일 한개의 구동기가 고장나면 모든 자료를 재구축하는데 기우성정보가 리용된다. 또한 두개 장치가 고장나면 모든 자료를 잃어 버리게 된다.

- 이 준위가 자주 리용되지 않는 리유는 기우성정보가 한개 디스크에 보존되기때문이다. 이 정보는 다른 디스크들중의 다른 어느 하나에 기록될 때마다 매번 갱신되여야 한다. 따라서 기우성디스크가 다른 디스크들보다 속도가 훨씬 뜨기때문에《병목》처럼 되여 버린다. 하지만 여러개의 속도가 뜬 디스크와 함께 속도가 빠른 디스크를 한개 가지게 되는 경우에 이 RAID준위는 아주 쓸모가 있다.

- RAID-5

  이 준위에 RAID가 많은 물리적디스크들을 결합하려고 할 때 제일 쓸모 있는 RAID방식으로 되며 여전히 일정한 여유도를 가지고 있다. RAID-5는 령 혹은 그 이상의 예비디스크들을 가진 3개 혹은 그이상의 디스크들에 리용할수 있다. RAID-5의 장치크기는 RAID-4와 같이 (N–1)×S로 된다. RAID-5와 RAID-4의 차이는 RAID-4에서 제기된《병목》현상을 없애기 위하여 포함된 디스크들속에 기우성정보가 골고루 분배된다는데 있다.

- 만약 디스크들중의 하나가 고장났을 때 여전히 모든 자료가 그대로 보존되는것은 기우성정보가 우월하기때문이다. 예비디스크들을 리용할수 있으면 장치가 고장난후에 즉시 재구축을 시작할수 있다. 두개 디스크가 동시에 고장나면 모든 자료를 잃어 버리게 된다. RAID-5는 한개 디스크고장은 회복할수 있으나 두개이상에 대하여서는 불가능하다.

▲ 읽기와 쓰기성능은 일반적으로 높아 지지만 어느 정도인가를 예측하기는 어렵다.

2.3.1. 예비디스크

예비디스크는 동작중에 있는 어느 하나의 디스크가 고장날 때까지도 RAID모임에 포함되지 않는 디스크를 의미한다. 어느 한 장치에 대하여 고장이 검출되면 그 장치는《불량》으로 표시되며 리용가능한 첫번째 예비디스크를 리용하여 즉시 재구축을 시작할수 있다. 따라서 예비디스크들은 물리적으로 실현하기는 좀 어렵지만 RAID-5체계에 대하여 특별히 좋은 여유안전성을 추가적으로 제공한다. 모든 여유도가 여유디스크에 의하여 보존되기때문에 이 방법은 고장난 장치가 있어도 아무때나 체계를 정상실행시킬수 있다. 사용자는 자기의 체계를 가지고 디스크폭주를 회복시킬수 없다. RAID계층은 장치적오유를 잘 조종해야 하지만 SCSI구동기들은 오유처리과정에 정지될수도 있고 또 IDE소편모임은 봉쇄되거나 기타 다른 일들도 생길수 있다.

2.4. RAID상에서의 교환기능

교환성능을 높이기 위하여 RAID를 리용할 근거는 하나도 없다. 핵심부는 자체로 여러개의 디스크상에서 교환작업을 분할할수 있다.

우수한 fstab은 다음과 같다.

/dev/sda2	Swap	swap	defaults, pri=1 00
/dev/sdb2	swap	swap	defaults, pri=1 00
/dev/sdc2	swap	swap	defaults, pri=1 00
/dev/sdd2	swap	swap	defaults, pri=1 00
/dev/sde2	swap	swap	defaults, pri=1 00
/dev/sdf2	swap	swap	defaults, pri=1 00
/dev/sdg2	swap	swap	defaults, pri=1 00

우의 설정은 기계가 7개의 SCSI구동기상에서 병렬로 교환을 진행할수 있게 한다. 이것은 오랜 기간 핵심부의 장점으로 되여 있었기때문에 RAID에 관하여서는 아무런 필요도 없다. 교환에 RAID를 리용하는 다른 리유는 그것의 높은 유용성이다. 사용자가 만일 체계를 기동할수 있게 실례로 RAID-1장치를 설정하면 체계는 디스크의 고장시 구조작업을 진행할수 있게 된다. 그러나 체계가 고장난 디스크상에서 교환되게 되면 틀림없이 실패하게 될것이다. RAID-1장치에서의 교환동작은 이 문제를 풀수 있게 한다. 교환이 RAID장치들우에서 안정하겠는가에 대하여서는 많이 론의되여 왔다. 또한 이 문제가 핵심부의 서로 다른 측면들에 강하게 의존하기때문에 계속 론의중에 있다. 이러한 내용들로 하여 RAID상에서의 교환동작은 배렬이 재구축될 때를 제외하고는 완전히 안정한것처럼 보인다(즉 새로운 디스크가 등급이 낮아 진 배렬에 삽입된후). 2.4에 와서 이 문제는 아주 뚜렷하고도 긴절하게 제기되였다. 하지만 그때까지도 사용자는 안정성을 만족시키거나 혹은 RAID상에서 교환기능을 할수 없다는 결론이 얻어 질 때까지 자체로 동작상태에서 체계검사를 진행하여야 하였다. 사용자는 자기의 RAID장치의 파일체계의 교환파일(swap file)에 RAID를 설치할수 있다. 한편 RAID장치를 교환구획으로 설정할수도 있다. 평상시와 같이 RAID장치는 곧 블로크장치이다.

3. 장치적문제

이 부분은 쏘프트웨어 RAID를 실행시킬 때 제기되는 몇가지 관련부분들을 언급한다.

3.1. IDE배치구성(configuration)

IDE디스크에서 RAID를 실제적으로 실행시킬수 있으며 좋은 성능을 얻을수 있다. 사실 IDE구동기들과 조종장치에 관한 문제는 새로운 RAID체계를 설정할 때 IDE에 관한 문제를 고찰하여야 한다.

▼ ***물리적안정성.*** IDE구동기들은 전통적으로 SCSI구동기들보다 공학적으로 볼 때 질이 더 낮다. 오늘에 와서도 IDE구동기들에 대한 품질담보는 1년이지만 SCSI구동기에 대하여서는 3년~5년이다. IDE구동기들에 대한 매 정의들이 불충분하게 주어 졌다고 말하기가 좀 공평하지 못하다 해도 일부 IDE구동기들이 류사한 SCSI구동기들보다 못하다는것을 알아야 한다. 그러나 일부 다른 구동기들은

SCSI와 IDE구동기들과 동일한 기계적설치로 리용할수 있다.

- ***자료통합성*** 이전에 IDE는 IDE모선에 보낸 자료가 디스크에 씌여 진 자료와 실제적으로 꼭 같다는것을 확인할만 한 방법이 없었다. 이 원인은 전체적인 기우성의 결핍, 검열결과가 같은것들때문이였다. 표준 Ultra-DMA에서 IDE구동기들은 현재 구동기가 접수하는 자료에 대하여 검열합을 취하며 따라서 자료가 손상되는 일은 없어 지게 된다.
- ***성능*** 여기서 IDE성능에 대하여 상세히 서술하려고 한다.
  IDE구동기들은 속도가 빠르다(12MB/s 혹은 그이상).
  IDE는 SCSI보다 CPU휴지시간이 더 크다.
  한개 IDE구동기에는 하나의 IDE모선만을 리용할수 있다. 따라서 종속디스크들은 성능이 떨어 진다.

▲ ***오유회복*** IDE구동기는 보통 IDE장치의 고장을 복구한다. RAID층은 고장난 디스크에 대하여 표식을 달며 RAID준위 1이나 그이상에서 실행되고 있으면 유지상태에서 해제될 때까지 작업하게 된다. IDE모선당 한개 IDE디스크를 리용할수 있다는 사실은 아주 중요하다. 이 사실은 두 디스크를 사용하지 못하게 할뿐아니라 흔히 디스크의 고장이 모선의 고장을 초래하며 따라서 모든 디스크의 고장은 그 모선상에서의 오유를 초래하게 된다. 고장안전 RAID설치(RAID준위 1, 4, 5)에서 디스크의 고장은 조종할수 있지만 두개 디스크의 고장은(한 디스크의 고장으로 하여 고장난 두 디스크) 배렬을 파괴할수 있다. 또한 모선우에서 기본구동기가 고장났을 때 종속구동기나 IDE조종장치는 몹시 혼란되게 된다. 그러므로 한개 모선에 한개의 구동기를 설치하는것은 하나의 규칙으로 된다. 이외에도 값이 눅은 PCI IDE조종장치들이 있다. SCSI디스크보다 더 눅은 IDE디스크들을 사용하면서 대표적인 체계들에도 추가할수 있는 디스크라는것을 알게 되였다. IDE는 큰 체계로 구성할 때 케블문제가 중요하게 제기된다. 사용자가 충분한 수의 PCI슬로트들을 가지고 있다고 해도 체계에 8개이상의 디스크들을 정합시킬수 없었지만 여전히 자료파손이 없이 실행되고 있다.

3.2. 활성교환

이 기능은 어떤 시각에 Linux핵심부목록에서의 활성문제이다. 구동기의 활성교환기능이 일부 범위들을 지원한다고 해도 여전히 사람이 쉽게 할수 있는것보다 못하다.

3.2.1. IDE구동기의 활성교환기능

IDE는 활성교환을 전혀 지원하지 못한다. 그래도 사용할수는 있으며 만일 사용자의 IDE구동기가 모듈로 콤파일되면 구동기에 재배치된후 그것을 다시 읽을수 있다. 그러나 고장난 IDE조종장치를 가지고 작업을 끌낼수도 있으나 그렇게 되면 해제된 체계에 구동기를 재배치하는데 걸린 시간보다 해제시간이 훨씬 더 길어 진다. 장치적으로 파괴될수 있는 전기적문제를 제외하고 기본문제는 IDE모선이 디스크가 교환된후에 다시 주사되여야 한다는것이다. 현재의 IDE구동기는 이 동작을 할수 없다. 만일 새로운 디스크가 변경

된 디스크와 100%같은것(무게, 기하학적크기 등)이라면 재주사를 하지 않고도 작업할수 있지만 실지로는 매우 위험하다.

3.2.2. SCSI구동기의 활성교환기능

표준 SCSI장치도 역시 활성교환을 지원하지 못한다. 그러나 이 장치로 작업은 할수 있다. 사용자의 SCSI구동기가 모선의 재주사, 구동기의 첨가 및 삭제기능을 지원하는 활성교환장치로 리용할수 있다. 그러나 표준 SCSI모선상에서 체계에 여전히 전원이 투입된 상태에서 장치의 플라그(plug)를 해제하는 동작은 하지 못할수 있다. 하지만 다시 해보면 동작할수도 있다. SCSI층은 디스크가 고장나면 원상태로 회복해야 하지만 모든 SCSI구동기는 아직 이 기능을 지원하지 못한다. 디스크가 해제될 때 SCSI구동기가 정지되면 체계는 이 상태를 계속 유지할수 있으며 활성플라그는 사실상 흥미가 없게 된다.

3.2.3. SCA에 의한 활성교환기능

SCA를 리용하여 장치를 활성플라그할수 있다. 하지만 이 기능을 수행하는 장치가 아직은 나오지 못하였다.

만일 사용자가 이 기능을 실현해 보려고 한다면 SCSI와 RAID에 대하여 파악해야 한다.

이와 관련하여 몇가지 문제점들을 제기한다.

▼ linux/drivers/scsi/scsi.c의 제거-단일-장치에 대하여 조사한다.

▲ raidhotremove와 raidhotadd를 찾는다.

모든 SCSI구동기들은 장치의 추가/삭제기능을 지원하지 못한다. 핵심부의 2.2계렬에서 적어도 Adaptec2940과 Symbios NCR53C8xx구동기가 이 기능을 지원할것 같고 다른것들은 할수도 있고 못할수도 있다.

4. RAID설정

4.1. 일반적설정

RAID설정은 임의의 RAID준위에 대하여 사용자에게 필요하다.

▼ ***핵심부*** 안정판 2.2.x핵심부 혹은 제일 후판 2.0.x

- **RAID*검사수정*** 보통 최근의 핵심부에 리용할수 있는 검사수정이다(2.4핵심부를 얻는다면 검사수정들은 그안에 이미 들어 있으며 따라서 사용자는 검사수정에 대하여 생각하지 않아도 된다.).

▲ **RAID*도구***

이 프로그람들은 모두 ftp : //ftp.fi. kernel. org/pub/linux에서 찾아 볼수 있다. RAID도구와 검사수정들은 /raid/alpha등록부의 데몬에 있다. 핵심부는 핵심부등록부에서 찾을수

있다.

핵심부를 검사수정하고 사용자가 쓰려고 하는 준위에 RAID지원부를 포함시키기 위하여 배치구성을 진행한다. 그다음 RAID도구의 압축을 해제하고 배치구성을 진행하며 콤파일하여 설치한다.

재기동할 때 사용자는 /proc/mdstat라는 파일을 가지고 있어야 한다. 다음 RAID모임에 포함시키려고 하는 구획을 생성한다.

4.2. 선형방식

사용자가 꼭 같은 크기가 아닌 두개이상의 구획을 가지고 있고 그것들을 서로 추가하려고 한다. 설치를 서술하기 위하여 /etc/raidtab를 설정한다. 선형방식으로 두개의 디스크에 관하여 raidtab를 설치한다. 파일형식은 다음과 같다.

```
raiddev /dev/md0
      raid-level           linear
      nr-raid-disks           2
      chunk-size             32
      persistent-superblock  1
      device              /dev/sdb6
      raid-disk               0
      device             /dev/sdc5
      raid-disk               1
```

예비디스크들은 여기서 지원되지 않는다. 만일 한개의 디스크가 파괴되면 그것과 함께 배렬도 파괴된다. 따라서 예비디스크에 보존할 정보도 없다. 배렬을 생성하기 위하여 다음의 명령을 수행한다.

```
mkraid/dev/md0
```

이 조작은 배렬을 초기화하고 영구상위블로크를 기록하며 다음 배렬을 출발시킨다. 이제 /proc/mdstat를 보면 배렬이 실행되고 있다는것을 알수 있게 된다. 이제 사용자는 임의의 다른 장치에서 생성한것과 같은 파일체계를 생성할수 있으며 그것을 올려태우기하고 사용자의 fstab나 기타 위치에 그것을 포함시킨다.

4.3. RAID-0

RAID-0은 대략적으로 같은 크기를 가진 장치들중에서 2개이상의 장치들의 기억용량을 병렬로 접근시켜 결합시키는 방법이다. 배치구성을 서술하기 위하여 /etc/raidtab파일을 설정한다. 실례를 아래에 보여 주었다.

```
raiddev              /dev/md0
raid-level              0
nr-raid-disks           2
persistent-superblock  1
chunk-size              4
device               /dev/sdb6
raid-disk               0
device              /dev/sdc5
raid-disk               1
```

여기에서도 역시 선형방식에서와 같이 예비디스크들을 지원하지 않는다. RAID−0은 여유도가 없으며 디스크가 파괴될 때 배렬도 파괴된다. 그러면 다시 배렬을 초기화하기 위하여

```
mkraid/dev/md0
```

을 실행한다. 이 조작은 상위블로크를 동기화하며 raid장치를 시동한다. 예비동작이 계속 진행되는가를 알아 보기 위하여 /proc/mdstat를 실행할수 있다. 그러면 현재 사용자의 장치가 실행되는가를 알수 있을것이다. /dev/md0은 양식화, 올려태우기준비가 된다.

4.4. RAID-1

사용자가 대략적으로 같은 크기를 가지는 2개의 장치를 가지고 있으면서 서로 거울로 되게 하려고 할 경우가 있다. 실제적으로 사용자는 보다 많은 장치를 가지고 있으며 그 장치를 보조대기(stand−by)예비디스크로 보존하려고 하면 그것은 능동디스크들중의 하나가 정지될 때 자동적으로 거울의 한 부분으로 될것이다. 그와 같은 /etc/raidtab파일을 설정한다.

```
raiddev/dev/md0
        raid-level       1
        nr-raid-disks    2
        nr-spare-disks   0
        chunk-size       4
        persistent-saperblock 1
        device /dev/sdb  6
        raid-disk        0
        device       /dev/sdc5
        raid-disk        1
```

만약 예비디스크를 가지고 있다면 그 장치들을 장치 정의부분의 끝에 추가할수 있다.

```
device     /dev/sdd5
spare-disk       0
```

이것은 일치하게 nr−spare−disks입구점들을 설정한다는데 대하여 강조해 둔다.
이제 RAID를 초기화하기 위하여 다음의 내용을 설정한다.

1. 거울이 구성되여야 한다. 즉 두개 장치의 내용(장치가 양식화되지 않았기때문에 아직 중요하지 않다.)이 동기화되여야 한다.
2. 거울초기화를 시작하기 위하여

```
mkraid/dev/md0
```

   명령을 내보낸다.
3. /proc/mdstat파일을 검사한다. 이 조작은 /dev/md0장치가 시동되였으며 거울이 재구성되고 있다는것 그리고 재구성의 완성에 대한 ETA를 사용자에게 알린다.
4. 재구성은 휴식중의 I/O대역폭을 리용하여 진행한다. 이때 체계는 현재 디스크 LED가 빨갛게 표시되였다 해도 여전히 뚜렷하게 응답하여야 한다.
5. 재구성과정은 명백하며 사용자는 현재 거울이 재구성상태에 있다고 해도 장치를 실제적으로 리용할수 있다.
6. 재구성이 진행되는동안 장치를 양식화한다. 이제부터는 작업할수 있다. 사용자는 장치를 올려태우기할수 있으며 재구성이 진행되는동안 그것을 리용할수 있다. 물론 재구성이 진행되는동안에 불량디스크가 중지되면 유감스러운 일로 된다.

4.5. RAID-4

주의 설치에 대한 다음의 내용은 어디까지나 추측에 불과하다는것을 지적해 둔다.

사용자는 대략적으로 같은 크기의 3개 혹은 그이상의 장치들을 가지고 있으며 그중 한개 장치는 다른 장치보다 현저하게 속도가 빠르다. 또한 이 장치들을 모두 몇가지 여유정보를 보존하는 하나의 큰 장치로 결합하려고 한다. 결국 사용자는 예비디스크로서 사용할 많은 장치들을 가지고 있다. 이 경우에 /etc/raidtab파일을 다음과 같이 설정한다.

```
raiddev  /dev/md0
    raid-level        4
    nr-raid-disks     4
    nr-spare-disks     0
    persistent-superblock 1
    chunk-size       32
    device           /dev/sdbl
    raid-disk        0
    device       /dev/sdcl
```

```
raid-disk        1
device       /dev/sdd1
raid-disk        2
device       /dev/sde1
raid-disk        3
```

만일 우리가 임의의 예비디스크를 가지고 있다면 그것들을 raid-disk정의에 따라 여느때와 같이 류사한 방법으로 삽입할수 있다.

```
device       /dev/sdf1
spare-disk       0
```

우리가 가지고 있는 디스크배렬은 mkraid/dev/md0명령을 리용하여 보통의 방법으로 초기화할수 있다.

사용자는 장치를 양식화하기전에 mke2fs에 관한 특정한 선택항목들에 대하여 알아야 한다.

4.6. RAID-5

사용자는 대략적으로 같은 크기를 가지는 3개 혹은 그이상의 장치들을 가지고 있으며 그것들을 하나의 큰 장치로 결합한다고 하지만 자료의 안정성을 위하여 어느 정도의 여유도를 유지하려고 한다. 한편 사용자는 다른 장치가 고장나기전에 배렬안에 포함시키지 않은 예비디스크로 리용하기 위하여 여러개의 장치들을 가지고 있다.

만일 사용자가 가장 작은 크기 S를 가지는 N개의 장치를 가지고 있다면 전체 배렬의 크기는 (N–1)×S로 될것이다. 이때 《흘려 버린》공간은 기우성정보(여유도)에 리용된다. 따라서 임의의 디스크가 고장나면 모든 자료가 그대로 유지된다. 하지만 두개의 디스크가 고장나면 모든 자료는 잃어 진다.

이를 위하여 /etc/raidtab파일을 다음과 같이 설정한다.

```
raiddev /dev/md0
        raid-level               5
        nr-raid-disks            7
        nr-spare-disks           0
        persistent-superblock    1
        parity-algorithm    left-symmetric
        chunk-size              32
        device              /dev/sda3
        raid-disk                0
        device              /dev/sdb1
```

```
raid-disk              1
device             /dev/sdc1
raid-disk              2
device             /dev/sdd1
raid-disk              3
device             /dev/sde1
raid-disk              4
device             /dev/sdf1
raid-disk              5
device             /dev/sdg1
raid-disk              6
```

만일 우리가 임의의 예비디스크를 가지고 있다면 raid디스크정의에 따라 류사한 방법으로 그것들을 삽입할수 있다.

```
device          /dev/sdh1
spare-disk          0
```

32KB의 덩어리(chunk)크기는 이 크기를 리용하는 많은 일반목적파일체계에서 좋은 기정값(default)으로 된다. 우에서 언급한 raidtab가 리용된 배렬은(7-1)×6GB=36GB를 가진 장치이다. 이 장치는 4KB블로크크기를 가지는 ext2fs파일체계를 유지한다. 만일 사용자의 파일체계가 더 크거나 혹은 매우 큰 파일들을 가지고 있다면 배렬의 덩어리크기와 파일체계의 블로크크기를 둘다 더 크게 취할수 있다. 충분히 이야기되였으므로 이제 raidtab를 설정하고 어떻게 동작하는가를 알아 보자.

```
mkraid/dev/md0
```

이 명령을 실행시키고 어떤 일이 생기는가를 보자. 기대했던대로 디스크들은 배렬의 재구성을 시작하면서 맹렬히 동작하기 시작한다. 어떤 동작이 계속되고 있는가를 알기 위하여 /proc/mdstat를 찾아 볼수 있다. 만일 장치가 성과적으로 생성되면 이제 재구성과정이 시작된다. 사용자의 배렬은 이 재구성단계가 완성될 때까지 일관성을 유지하지 못한다.

그러나 배렬이 충분하게 동작하면 그것을 양식화할수 있고 재구성되고 있는 동안에도 리용할수 있다. 배렬을 양식화하기전에 mke2fs의 특정한 선택사항들에 관한 부분을 보는것이 좋다.

사용자는 이제 RAID장치가 실행중에 있을 때 아무때나 그것을 정지시킬수 있고 혹은 다음의 명령

raidstop/dev/md0이나 혹은 raidstart/dev/md0을 리용하여 정지시킬수도 있고 재출발시킬수도 있다.

4.7. 영구상위블로크

이제 다시 앞으로 돌아 와서 raid도구로 /etc/raidtab파일을 읽고 배렬을 초기화한다. 하지만 이 조작은 /etc/raidtab가 존재하는 파일체계가 올려태우기될것을 요구한다. 사용자가 RAID로 기동하려고 하는 경우에는 이 방법이 좋은 방법으로 되지 못한다.

변경된 방법들은 RAID장치상에서 파일체계를 올려태우기할 때 복잡한 문제들을 야기시킨다. 그러한 파일체계들은 여느때처럼 /etc/fstab파일에 설치되지 않고 init−scripts부터 올려태우기되여야 한다.

영구상위블로크들은 이러한 문제점을 해결할수 있게 한다. 배렬이 /etc/raidtab파일에서 영구상위블로크선택항목으로 초기화될 때 특정한 상위블로크가 배렬에 포함되는 모든 디스크들의 앞머리에 씌여 진다. 이 과정은 핵심부로 하여금 늘 리용할수 없는 일부 배치구성파일로부터 읽기를 진행할 대신에 포함된 디스크들로부터 직접 RAID장치의 배치구성을 진행할수 있게 한다.

그러나 사용자는 배렬의 다음번 재구성을 위하여 그런 요구를 제기할수도 있기때문에 /etc/raidtab파일의 일관성이 여전히 유지되여야 한다.

영구상위블로크는 체계가 기동한 상태에서 RAID장치들을 자동검출할 때 그 관리기로 된다. 이 내용은 자동검출부분에서 설명한다.

4.8. 덩어리의 크기

덩어리크기는 설명해야 할 가치가 있는 내용이다. 사용자는 디스크들의 모임에 대하여 완전한 의미에서 병렬쓰기를 진행할수 없다. 만약 사용자가 두개의 디스크를 가지고 있으면서 한개 바이트를 쓰려고 한다면 매개 디스크에 4개의 비트를 써야 할것이다. 실제적으로 매개 두번째 비트는 디스크 0에 가야 하고 다른것들은 디스크 1에 가야 한다. 장치로써는 이 과정을 지원하지 못한다. 대신 우리는 임의로 덩어리크기를 선택할수 있는데 이 덩어리크기는 장치에 쓸수 있는 자료의 제일 작은 《원자적》모임으로 정의한다. 4KB의 덩어리크기를 16KB에 쓰는 조작은 두개 디스크를 가지는 RAID−0의 경우에 첫 4KB덩어리와 세번째 4KB덩어리를 첫번째 디스크에 쓰며 두번째와 네번째 덩어리는 두번째 디스크에 기록하게 한다. 따라서 대량쓰기에서는 될수록 큰 덩어리들을 리용하면 휴지시간을 감소시킬수 있으며 따라서 작은 파일들을 가지고 있는 배렬들은 보다 작은 덩어리크기로부터 보다 큰 리익을 얻을수 있다.

▼ 덩어리크기들은 선형방식을 포함하여 모든 RAID준위들에서 정의되여야 한다.
- 성능을 확고히 높이기 위하여 값들만이 아니라 배렬우에 적재된 파일체계의 블로크크기를 가지고도 실험을 해야 한다.

▲ /etc/raidtab에서 덩어리크기에 대한 인수는 KB단위로 정의되는데 《4》는 4KB를 의미한다.

4.8.1. RAID-0

자료는 배렬안의 디스크들에 《거의》병렬로 씌여 진다. 덩어리크기를 4KB로 정의하고 3개의 디스크배렬에 16KB를 쓰면 RAID체계는 디스크 0, 디스크 1, 디스크 2에 병렬로 4KB를 쓰고 나머지 4KB는 디스크 0에 쓴다.

32KB로 덩어리크기를 정의하는것이 대다수의 배렬들에서 적합한 출발점으로 되고 있다. 하지만 이 현저한 값은 포함된 구동기의 수, 거기에 설정된 파일체계의 내용, 기타 다른 요소들에 크게 의존하게 된다. 그러므로 이 값이 더 좋은 성능을 얻으려면 실험해 보아야 한다.

4.8.2. RAID-1

쓰기과정에서는 덩어리의 크기가 배렬에 아무런 영향도 주지 않는다. 왜냐하면 모든 자료가 아무런 장애도 없이 써지기때문이다. 그러나 읽기에 대해서는 덩어리크기가 포함되는 디스크들로부터 얼마만한 자료를 직렬로 읽는가를 정의한다. 배렬안의 모든 능동디스크들이 같은 정보를 포함하기때문에 읽기는 RAID−0에서와 같이 병렬로 진행될수 있다.

4.8.3. RAID-4

쓰기가 RAID−4배렬에서 진행될 때 기우성비트는 기우성디스크상에서도 갱신되여야 한다. 이때 덩어리크기는 기우성블로크들의 크기이다. 만일 한개바이트가 RAID−4배렬에 기록되면 덩어리크기바이트들은 N−1개의 디스크들로부터 읽을수 있으며 기우성정보가 계산되고 덩어리크기바이트들이 기우성디스크에 기록된다.

덩어리크기는 RAID−0에서와 꼭같은 방법으로 읽기성능에 영향을 준다.

4.8.4. RAID-5

RAID−5에서 덩어리크기는 RAID−4와 완전히 꼭같은 의미를 가지고 있다. RAID−5에 적합한 덩어리크기는 128KB이지만 이 크기로 실험을 해보려는것도 좋다.

또한 mk2fs에 관한 특정한 선택항목에서 이 부분을 고찰해 보아야 한다. 이 내용이 RAID−5의 성능에 영향을 준다.

4.9. mke2fs의 선택항목들

mke2fs로 RAID−4와 RAID−5를 양식화할 때 리용할수 있는 특정한 선택항목이 있다. −R stride=nn항목은 RAID장치상에서 지능적인 방법으로 mke2fs를 ext2정의형자료구조와 다르게 배치할수 있게 한다.

덩어리크기가 32KB이면 32KB의 련속적인 자료가 디스크우에 존재하게 된다. 우리가 4KB의 블로크크기를 가지고 ext2파일체계를 구성하려고 한다면 한개배렬덩어리에 8개의 파일체계블로크를 구성할수 있을것이다. 우리는 파일체계를 생성할 때 이 정보를 mk2fs

편의프로그람에 넘겨 줄수 있다.

```
mke2fs-b 4096 -R  stride=8  /dev/md0
```

RAID−{4,5}의 성능은 엄격히 이 선택항목의 영향을 받는다. 이때 분할결음선택항목이 다른 RAID준위들에 어떤 영향을 줄것인가에 대하여서는 확신하지 못하고 있다.

ext2fs블로크크기는 파일체계의 성능에 큰 영향을 준다. 사용자는 항상 파일체계안에 매우 작은 파일들을 대량적으로 기억시키지 않는 한 수백메가바이트이상의 임의의 파일체계에 대해서는 4KB의 블로크크기를 사용해야 한다.

4.1 0. 자동검출

자동검출은 기동시에 일반 구획검출이 끝난 다음 즉시 핵심부에 의하여 RAID장치들이 자동적으로 인식되게 한다. 이 조작은 몇가지 조건들을 요구한다.

1. 핵심부안에서 자동검출지원기능이 필요하다. 이것을 검사해야 한다.
2. 영구상위블로크를 리용하여 RAID장치들을 생성해야 한다.
3. RAID에 리용된 구획형들은 OxFD(fdisk를 리용하여 형을 “fd”로 설정)로 설정해야 한다.

주의 : 사용자의 RAID가 구획형이 발견되기전에 실행되지 않는다는것을 알아야 한다. 장치를 정지시키기 위하여 raidstop/dev/md0를 리용할수 있다.

만일 사용자가 우로부터 1, 2, 3을 설정하면 자동검출이 설정될것이다. 재기동하고 체계가 동작되기 시작할 때 /proc/mdstat로 RAID가 실행중에 있다는것을 통보하여야 한다. 기동중에 사용자는 아래와 류사한 통보문을 볼수 있다.

```
oct 22 00:51:59 malthe kernel:SCSI device sdg:hdwrsector=512
  bytes, Sectors=12657717 [6180MB] [6.2GB]
oct 22 00:51:59 malthe kernel: partition check:
oct 22 00:51:59 malthe kernel: sda: sda1 sda2 sda3 sda4
oct 22 00:51:59 malthe kernel: sdb: sdb1 sdb2
oct 22 00:51:59 malthe kernel: sdc: sdc1 sdc2
oct 22 00:51:59 malthe kernel: sdd: sdd1 sdd2
oct 22 00:51:59 malthe kernel: sde: sde1 sde2
oct 22 00:51:59 malthe kernel: sdf: sdf1 sdf2
oct 22 00:51:59 malthe kernel: sdg: sdg1 sdg2
oct 22 00:51:59 malthe kernel: autodetecting RAID arrays
oct 22 00:51:59 malthe kernel: (read) sdb1’s sb offset: 619872|
oct 22 00:51:59 malthe kernel: bind
oct 22 00:51:59 malthe kernel: (read) sdc1’s sb offset: 619872|
```

```
oct 22 00:51:59 malthe kernel: bind
oct 22 00:51:59 malthe kernel: (read) sdd1’s sb offset: 619872|
oct 22 00:51:59 malthe kernel: bind
oct 22 00:51:59 malthe kernel: (read) sdel’s sb offset: 619872|
oct 22 00:51:59 malthe kernel: bind
oct 22 00:51:59 malthe kernel: (read) sdfl’s sb offset: 6205376
oct 22 00:51:59 malthe kernel: bind
oct 22 00:51:59 malthe kernel: (read) sdg1’s sb offset: 6205376
oct 22 00:51:59 malthe kernel: bind
oct 22 00:51:59 malthe kernel: autorunning md0
oct 22 00:51:59 malthe kernel: running
oct 22 00:51:59 malthe kernel: now!
oct 22 00:51:59 malthe kernel: md: md0: raid array is not clean-
  starting background reconstruction
```

이 통보문은 명백하게 전원이 꺼지지 않은 RAID−5의 자동검출과정에 출력되는 통보문이다(실례로 체계가 폭주된 경우). 재구성은 자동적으로 시동된다. 이 장치의 올려태우기조작은 재구성이 명백하고 모든 자료의 일관성이 보장되므로 안정성이 완전히 보장된다(기우성정보망은 일관성이 없지만 장치가 고장나기전에는 필요없다.).

자동출발한 장치들은 전원차단시 역시 자동적으로 정지된다.

init scripts는 걱정하지 않아도 된다. 이제 다른 명령으로서 /der/md를 사용하자.

```
/dev/sd or /dev/hd devices
```

사용자는 임의의 raidstart/raidstop명령에 관한 init-scripts를 보고싶을수 있는데 이 정보들은 흔히 표준 RedHat init scripts에서 찾아 볼수 있다. 이 내용은 변경된 형식의 RAID에 리용되며 자동검출을 지원하는 새 형식의 RAID에는 없다.

4.1 1. RAID상에서의 기동

RAID장치상에서 뿌리파일체계를 올려태우기하는 체계를 설정하는데는 여러가지 방법이 있다. 어떤 때는 RedHatLinux 6.1의 그라프설치프로그람만이 RAID장치를 직접적으로 설정하게 한다. 이렇게 하고 싶을 때 많은 사람들은 체계의 기능변경을 하여야 할것처럼 인식되며 가능하다. 가장 최근의 사무용lilo배포판(판본 21)은 RAID를 조종하지 못하며 따라서 핵심부는 기동시 RAID장치로부터 적재될수 없다. 사용자가 이 판본을 리용하면 /boot파일체계는 비RAID장치에 존재하여야 한다. 체계가 기동한다고 해도 그것을 담보하는 방법은 사용자의 RAID의 모든 구동기에 류사한 /boot구획이 생성하는것이다. 즉 BIOS가 어떤 장치 실례로 리용가능한 첫번째 구동기로부터 자료를 적재하게 하는 방법이다. 이 조작은 사용체계안의 고장난 디스크로 기동하지 말것을 요구한다.

redHat 6.1에서 1 : lilo 21에 대한 검사수정은 RAID−1에서 조종하거나 기동할수 있

다. 이 검사수정이 그 어떤 다른 순위에서 동작하지 않는다는것을 알아 두어야 한다. RAID−1(거울디스크)이 유일체계지원되는 RAID준위이다.

이 검사수정(lilo.raid1)은 임의의 reahat거울상의 dist/redhat−6.1/SRPMS/SRPMS/lilo−0.21−1.0.src.rpm에서 찾을수 있다. LILO의 검사수정판본은 lilo.conf에서 boot=/dev/md0을 받아 들이게 되며 매 디스크를 기동가능한 거울로 만들수 있게 한다.

사용자의 체계가 항상 기동할수 있다는것을 담보하는 또 다른 방법은 모든 설치를 진행하는 기동용플로피디스크를 만드는것이다. /boot파일체계에 존재하는 디스크가 고장난 디스크라면 언제나 플로피디스크로부터 기동할수 있다.

4.12. RAID상의 뿌리파일체계

RAID상에서 기동하는 체계를 준비하기 위하여서는 RAID장치상에 뿌리파일체계(/)가 올려태우기되여야 한다. 이 목적을 달성하기 위한 두가지 방법을 아래에 제공한다. 현행 배포판(적어도 알려 진것들중에서)에는 RAID장치에 설치하는 지원기능이 없기때문에 이 방법들은 표준구획상에서 설치한다고 가정하고 RAID가 아닌 장치의 뿌리파일체계를 새로운 RAID장치로 옮긴다.

4.12.1. 조작 1

이 방법은 사용자가 체계를 설치할수 있는 예비디스크를 가지고 있다고 가정하는데 이 디스크는 배치구성을 진행할 RAID장치가 아니다.

▼ 첫째로 예비디스크에 표준체계를 설치한다.

- 운영하려고 계획하는 핵심부, raid−검사수정, 도구들을 선택하고 새로운 RAID−인식핵심부로 체계를 기동한다. RAID−지원프로그람이 핵심부안에 있는가, 모듈로서 적재되지 않았는가를 확인한다.
- 다음 배치구성을 진행하고 뿌리파일체계로 쓰려고 하는 RAID를 생성한다. 이 조작은 이 책의 다른 부분에서 서술되고 있는것처럼 표준적인 절차로 된다.
- 매개 조작들이 원만히 되였다는것과 새 RAID가 기동상태로 준비되였는가를 알아 보기 위하여 체계를 재기동시킨다.
- 파일체계를 새로운 디스크배렬에 배치하고 (mke2fs를 리용하여) 그것을 mnt/newroot 아래에 올려태우기한다.
- 사용자의 현재 뿌리파일체계(예비디스크)의 내용을 새 뿌리파일체계(배렬)에 복사한다. 이 과정을 실현하는 방법들에는 여러가지가 있는데 우의것은 그것들중의 한가지 방법이다.

```
cd/
find. -xdev/cpio -pm/mnt/newroot
```

- 뿌리파일체계에 정확한 장치를 리용하기 위하여 /mnt/newroot/etc/fsstab파일을 변

경시켜야 한다.

- 현행 /boot 파일체계의 올려태우기를 해제하고 /mnt/newroot/root에 기동장치를 대신 올려태우기한다.
- 정확한 장치를 지적하기 위하여 /mnt/newroot/etc/lilo.conf를 갱신한다. 기동장치는 여전히 정규디스크여야 하지만 뿌리장치는 새로운 RAID를 지적해야 한다. 다 되면 오유없이 완성하는 lilo−r/mnt/newroot를 실행한다.

▲ 체계가 재기동하며 모든 상태가 예견했던대로 나타난다. 사용자가 이 조작을 IDE디스크들을 가지고 실행할 때는 모든 디스크들이 《자동검출》형식으로 되였는가와 BIOS에 통보하였는가를 알아야 하며 따라서 BIOS는 부정확한 디스크를 리용해도 체계를 기동시킬수 있게 될것이다.

4.12.2. 조작 2

이 방법은 사용자가 부정확한 디스크지령을 가지고 있는 raidtools.patch를 사용하는 경우에 적용하는 방법이다. 이 방법은 2.2.10과 그 이후의 모든 핵심부에 관한 도구/검사수정에 해당한다. 사용자는 이 방법을 RAID 1준위와 그 웃준위들에서만 리용할수 있다.

기본착상은 RAID에 틀린것(고장난것)으로 표기된 디스크에 체계를 설치하고 그다음 하급(degraded)방식으로 운영할 RAID에 체계를 복사하며 마지막에 하급방식이 아닌 상태에서 RAID를 운영하던 변경된 설치판을 지우고 RAID가 더이상 설치디스크가 없이도 운영되게 하는것이다.

▼ 우선 표준체계를 한개 디스크(후에 RAID부분으로 될)에 설치한다. 이 디스크(혹은 구획)가 제일 작은 디스크가 아니라는 점이 중요하다. 만일 제일 작은 디스크이면 그것을 RAID에 추가하는것은 불가능하다.

- 다음 핵심부, 검사수정, 도구 등을 선택한다. 사용자가 요구하는 RAID지원기능을 가지며 핵심부로 콤파일된 새로운 핵심부로 체계를 기동한다.
- raidtab파일의 고장난 디스크로서 현행뿌리장치를 가진 RAID를 설치한다. 고장난 디스크를 raidtab안의 첫번째 디스크로서 배치하지 말아야 한다. 이렇게 되면 RAID로서 출발하는데 일정한 문제가 생길수 있다. RAID를 생성하고 거기에 파일체계를 놓는다.
- 재기동하고 RAID가 정확히 동작하는가를 알아 본다.
- 앞부분에서 서술한것처럼 체계를 복사하고 뿌리장치로 RAID를 리용할수 있게 체계의 배치구성을 다시 한다.
- 체계가 RAID로부터 성과적으로 기동하면 고장난 디스크를 포함하는 raidtab파일을 표준 raid디스크로 변경할수 있다.

▲ 사용자는 하급상태가 아닌 RAID로부터 기동할수 있는 체계를 가질수 있게 되였다.

4.13. RAID에서 체계기동파일만들기

뿌리파일체계를 올려태우기할수 있는 핵심부에서 뿌리파일체계가 존재하는 장치에 대한 모든 지원기능이 핵심부에 있어야 한다.

따라서 RAID장치에 뿌리파일체계를 올려태우기하기 위하여서는 핵심부가 RAID지원기능을 가지고 있어야 한다. 핵심부가 RAID장치를 알고 있다는것을 확인하는 표준적인 방법은 자체안에 콤파일된 모든 필요한 RAID지원기능을 가진 핵심부를 콤파일하는것이다. RAID지원기능을 적재가능한 모듈로서가 아니라 핵심부로 콤파일한다는데 대하여 주의를 돌려야 한다. 핵심부는 뿌리파일체계가 올려태우기되기전에는 모듈을(뿌리파일체계로부터) 적재할수 없다.

하지만 raidHat−6.0의 새로운 형식의 RAID지원기능을 모듈로서 가지고 있는 핵심부로 배포되기때문에 여기서 표준 RaidHat−6.0핵심부를 어떻게 리용하는가에 대하여 서술한다.

4.13.1. RAID의 모듈로서의 기동

사용자는 이 조작을 수행하기 위하여 LILO에 RAM디스크를 사용할수 있게 지령을 주어야 한다.

뿌리구획을 올려태우기하기 위하여 필요한 모든 핵심부모듈을 포함하는 RAM디스크를 생성할수 있도록 mkinitrd명령을 리용한다. 이 조작은 다음과 같이 실행할수 있다.

Mkinitrd – – with=

실례로

mkinitrd – – with=raid5 raid−ramdisk2.2.5−22

이 조작은 뿌리장치를 올려태우기할 때 리용하기 위한 핵심부에 대하여 정의된 RAID모듈이 기동시에 존재하는가를 확인하게 될것이다.

4.14. 위 험

실행중에 있는 RAID의 구성부분인 디스크의 구획은 다시 설정할수 없다. 사용자가 RAID의 한 부분인 어느 한 디스크의 구획들을 교체하여야 한다면 우선 배렬의 동작을 중지시키고 다음에 구획의 재설정을 진행하여야 한다.

하나의 모선우에 여러개의 디스크를 설정하는것은 어렵지 않다.

표준고속–광대역 SCSI모선은 현재 많은 디스크들이 단독으로 실행할수 있는것보다 좀 더 작은 10MB/S의 속도를 유지하고 있다. 모선상에 이러한 디스크 6개를 설치하는것은 물론 사용자가 기대했던 정도의 성능을 보장하지 못한다. SCSI모선들이 디스크들을 서로 극단적으로 가깝게 배치하였을 때 보다 많은 SCSI조종장치들이 사용자에게 여유있는 성능을 제공할수도 있다. 한개의 조종장치에서 두개의 디스크를 실행시킬 대신에 두개의 변경된 SCSI디스크를 가진 두개의 2940형을 사용해서는 성능을 향상시킬수 없

을것이다.

만약 사용자가 영구상위블로크의 선택항목을 잊었을 때는 배렬이 정지된 이후에 그것을 다시 출발시킬수 없을수도 있다.

이때는 raidtab에 정확하게 설정된 선택항목으로 배렬을 다시 생성한다.

만일 한 디스크가 제거되였거나 혹은 삽입된후에 RAID-5가 재구성에서 실패하면 그것은 raidtab안의 디스크들의 순차성때문일수도 있다. 이때는 첫번째 "device…"과 "raid-disk"의 쌍을 raidtab파일의 배렬서술자 밑바닥으로 옮겨 본다.

우리가 Linux핵심부에서 볼수 있는 대부분의 《오유보고》는 정확한 판본의 raid도구들로 정확한 RAID-검사수정을 사용하기 위하여 무엇인가 해보다가 실패한 사람들이 제기한 내용이다. 사용자가 0.90RAID를 리용하고 있으면 그에 해당한 raidtools를 리용하고 있는가를 확인해야 한다.

5. 시 험

사용자가 고장견딤성을 얻기 위하여 RAID를 사용하려고 하면 그것이 실지적으로 동작하는가를 알기 위하여 설정을 시험해 볼 필요도 있을수 있다. 그러면 디스크의 결함을 어떻게 모의할수 있는가를 보자. 구동기가 고장일 때 어떤 일이 발생하는가에 대하여 누구도 알수 없다. 고장은 모선자체일수도 있고 전기적원인에 의하여 생길수도 있다. 구동기는 SCSI/IDE계층에 대한 읽기/쓰기오유에 대하여 통보할수도 있는데 이때 SCSI/IDE는 RAID계층이 이 상태를 깨끗하게 조종할수 있게 한다.

5.1. 구동기오유의 모의

구동기오유에 대하여 모의하려고 할 때 먼저 구동기의 플라그를 해제한다. 이것은 전원을 끄는 방법으로도 할수 있다. 사용자가 일반적인 개수보다 더 적은 디스크로써 자료를 회복할수 있겠는가를 검사하는데 흥미를 가진다면 그런 관점은 무모하고 어리석다. 체계를 개방하고 디스크의 플라그를 해제하여 다시 기동한다.

체계의 가동일지를 들여다 보고 RAID의 동작여부를 알아 보기 위해 /proc/mdstat를 고찰한다. 사용자는 자기의 디스크배렬의 디스크오유를 되살릴수 있도록 RAID-{1, 4, 5}의 운영상태에 있어야 한다는것을 상기해야 한다.

선형방식이나 혹은 RAID-0방식은 장치가 부정확하면 완전한 실패로 된다. 사용자가 디스크를 다시 련결할 때(물론 전원을 끈 상태에서)는 raidhotadd명령으로 RAID에 다시 새 장치를 추가할수 있다.

5.2. 자료손상모의

RAID(장치나 혹은 프로그람)는 만일 디스크에 대한 쓰기가 오유를 귀환시키지 않으면 쓰기가 성공하였다고 가정한다. 따라서 사용자의 디스크가 오유를 귀환시키지 않았는데 자료가 흐트러 지면 그 자료는 파괴된것으로 인정된다. 이러한 경우는 물론 생길것 같

지 않지만 가능하며 이것은 파일체계가 손상된 결과라고 볼수 있다.

RAID는 매체에서의 자료손상에 대하여 보호하는 문제는 제기하지 않고 있으며 할수도 없다. 그러므로 RAID체계가 오유를 어떻게 조종하는가를 알아 보기 위하여 일부러 자료를 손상시키는것 역시 리치에 맞지 않는다.

RAID계층이 자료손상에 대하여 찾을수 없다는것은 거의 명백하며 게다가 RAID장치에 있는 파일체계도 손상될수 있다. 이것은 연구과정에 제기된 방법의 하나이다.

RAID는 자료통합성을 담보하지 못하며 항상 디스크가 죽으면 자료를 보존할수 있게 한다(즉 같거나 높은 RAID준위에서).

6. 재 구 축

사용자가 이 참고서의 마지막부분을 읽어 보면 하급방식의 RAID의 재구축이 어떤 내용을 포함하는가에 대하여 잘 알게 될것이다.

▼ 체계의 전원정지
- 고장난 디스크의 교체
- 다시 체계의 전원투입
- 배렬에 디스크를 삽입하기 위하여 raidhotadd기능인 /dev/mdx/dev/sdx를 리용

▲ 자동재구축이 실행되는 일정한 시간 기다린다.

이 과정은 보통 사용자에게 별다른 일이 없고 또 여유 있는 디스크보다 더 많은 디스크가 고장난것으로 하여 RAID를 리용할수 있는 조건에서는 일반적으로 실행된다. 그러한 현상은 같은 모선상에 많은 디스크들이 존재하며 한개 디스크가 폭주된 디스크의 모선을 점유할 때 발생한다. 그러나 다른 디스크들은 모선이 내려 지기때문에 RAID계층에 도달되지 못할수 있으며 이때 그것들은 오유가 생긴것으로 표식된다.

사용자가 한개 예비디스크로 쓸수 있는 RAID−5에 관하여 두개 혹은 그이상의 디스크들을 풀어 놓는 조작은 치명적오유를 발생시킬수 있다.

6.1. 다중디스크오유의 회복

절차는 다음과 같다.

▼ 조종장치가 정지되고 동시에 두개 디스크들의 련결이 해제된다.
- 한개 SCSI모선의 모든 디스크들은 한개 디스크가 정지되면 더이상 도달될수 없다.

▲ 케블이 풀려 진다.

간단히 말하여 한번에 여러개의 디스크들에서 림시적고장이 생기는 경우가 있다.

후에 RAID상위블로크들은 동기에서 제외되며 사용자는 더이상 RAID배렬을 초기화할수 없다. 다른 하나의것은 mkraid에 의하여 상위블로크를 재쓰기한다. 만일 이것이 장치들과 동작이 진행되지 않는 초기디스크들의 순서를 일치시키지 못하면 이 조작을 실현하기 위하여 현재까지의 /etc/raidtab를 가지고 있어야 할것이다.

배렬을 시작하여 생성된 체계가동일지들을 찾아 보고 매개 상위블로크에 해당한 사건계수값을 파악한다.

만일 사용자가 고장난 디스크가 없이 mkraid를 실행하면 회복스레드가 즉시 얻어 지고 기우성블로크의 재구축이 시작된다.

7. 성 능

이 부분은 프로그람 RAID를 리용하는 실체계들로부터 제기되는 많은 성능평가기준을 포함한다. 성능평가는 시험프로그람에 의하여 수행되며 모든 시간동안에 파일에 관하여 두번 혹은 콤퓨터의 물리적RAM의 크기에 따라 그이상 진행할수도 있다. 여기에서의 성능평가는 하나의 대규모단일파일에 관하여 입력과 출력의 대역폭만을 측정한다. 알아야 할 문제가 하나 있는데 만일 큰 읽기/쓰기에 대하여 최대 I/O처리능력이 주어 지면 거기에 관심을 돌리게 된다. 그러나 이러한 수값들은 우리에게 배렬이 새로운 풀이나 Web봉사기 등에 리용될 때 어떤 성능이 발휘되겠는가에 대하여서는 극히 적은 정보만을 제공해 준다.

항상 성능평가기준값들이 《합성》프로그람의 실행결과라는데 대하여 알아야 한다.

특정한 콤퓨터에 대한 평가시험결과를 제시한다.

▼ 2중pentium pro 150MHZ
- 256MB RAM(60MHZ EDO)
- 3개의 IBM UltraStar 9ES 4.5GB,SCSI U2W
- Adaptee 2940 U2W
- 하나의 IBM UltraStar 9ES 4.5GB,SCSI UW
- Adaotec 2940 UW

▲ RAID patch를 가진 kernel 2.2.7

3개의 U2W디스크는 U2W조종장치에서 분리되며 UW디스크는 UW조종장치에서 분리된다.

RAID를 사용하거나 혹은 사용하지 않으면서 이 체계의 SCSI모선을 거쳐 30MB/s이상의 속도를 얻는것은 불가능한것처럼 보인다. 추측에 의하면 이 체계가 낡았기때문에 기억대역폭이 흡수되고 따라서 SCSI조종장치를 거쳐 보내지므로 속도에 제한이 있다는 것이다.

7.1. RAID-0

읽기는 직렬블로크들의 입력이며 쓰기는 직렬블로크들의 출력이다. 파일크기는 전체 시험과정에 1GB였다. 시험은 단일사용자방식으로 진행되였다. SCSI구동기는 배치구성되였으나 표식 붙은 배렬대기렬에는 리용하지 않는다.

덩어리크기(KB)	블로크크기(KB)	읽기(KB/s)	쓰기(KB/s)
4	1	19712	18035
4	4	34048	27061
8	1	19301	18091
8	4	33920	27118
16	1	19330	18179
16	2	28161	23682
16	4	33990	27229
32	1	19251	18194
32	4	34071	26976

이로부터 RAID덩어리크기에 의하여서는 차이가 크게 나지 않는것 같이 보인다. 그러나 ext2fs블로크크기는 가능한 정도로 커야 하며 이 값은 IA-32상에서 4KB(즉 페지크기)가 좋다.

7.2. TCQ를 가진 RAID-0

여기서는 SCSI구동기가 표식 붙은 명령대기렬을 사용하여 배치구성하는데 대기렬의 깊이는 8이다. 만일 그렇지 않으면 나머지 모든것은 앞의 경우와 꼭 같다.

덩어리크기	블로크크기	읽기 (KB/s)	쓰기(KB/s)
32k	4k	33617	27215

이상의 검사는 진행되지 않았다. TCQ에서는 쓰기성능이 약간 개선되여 보이지만 실지로는 전혀 큰 차이가 없다.

7.3. RAID-5

배렬은 RAID-5방식으로 실행할수 있도록 배치구성되였으며 거의 같은 검사들이 진행되였다.

덩어리크기	블로크크기	읽기 (KB/s)	쓰기 (KB/s)
8k	1k	11090	6874
8k	4k	13474	12229
32k	1k	11442	8291
32k	2k	16089	10926
32k	4k	18724	12627

덩어리크기와 블로크크기는 둘다 실제적인 차이가 있다.

7.4. RAID-10

RAID-10《거울화된 분할걸음》이거나 혹은 두개의 RAID-0배렬로 구성된 RAID-1배렬이다. 덩어리의 크기는 RAID-1배렬과 두개의 RAID-0배렬덩어리크기이다.

설정이 제대로 되였다 하더라도 덩어리크기가 다르면 검사하기가 곤난하다.

덩어리크기	블로크크기	읽기(KB/s)	쓰기(KB/s)
32K	1K	13753	11580
32K	4K	23432	22249

파일크기는 900MB였다. 왜냐하면 포함된 4개의 구획들의 매개가 500MB이고 이것들은 프로그람의 설치에 1GB대역을 주지 않기때문이다(두개의 1000MB배렬의 RAID-1).

부록 2. 참고문헌

Blair, David C. and Marron, M. E. "Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System," *Communications of the ACM*, v28 n3 Mar 1985, p. 289-299.

Codd, E. F. "The Relational Model for Database Management: version 2" c1990 AddisonWesley Pub. Co, Not recommended as a textbook, Date's is better for that, but worthwhile if you want a long paper by Codd. Notice that he places greater emphasis on closure, and design methodology principles in general, than designers of other naming systems such as hypertext.

Date, C.J. *An Introduction to Database Systems*, 4th ed. Reading, Mass.: Addison-Wesley Pub. Co., c1986. Contains a well written substantive textbook sneer at the problems of hierarchical naming systems, and a well annotated bibliography.

Curtis, Ronald and Wittie, Larry. "Global Naming in Distributed Systems," *IEEE Software*, July 1984, p. 76-80.

Feldman, Jerome A., Fanty, Mark A., Goddard, Nigel H. and Lynne, Kenton J. "Computing with Structured Connectionist Network,." *Communications of the ACM*, v31, Feb 1988, p. 170(18).

Fox, E. A., and Wu, H. "Extended Boolean Information Retrieval," *Communications of the ACM*, 26, 1983, pp. 1022-1036.

Gallant, Stephen I., "Connectionist Expert Systems," *Communications of the ACM*, v31 Feb 1988, p152(18).

Gates, Bill. Comdex '91 speech on "Information at Your Fingertips," available for $8 on videotape from Microsoft's sales department.

Gifford, David K., Jouvelot, Pierre, Sheldon, Mark A., O'Toole, James W. Jr., "Semantic File Systems," *Operating Systems Review*, v25, n5, October 13-16, 1991. They demonstrated that extending Unix file semantics to include nonhierarchical features is useful and feasible. Unfortunately, their naming system lacks closure.

Gilula, Mikhail. *The Set Model for Database and Information Systems*, 1st Edition, c 1994, Addison-Wesley. Provides a Set Theoretic Database Model in which relational algebra is a shown to be a special case of a more general and powerful set theoretic approach.

Joint Object Services Submission (JOSS), OMG TC Document 93.5.1.

Marchionini, Gary, and Shneiderman, Ben. "Finding Facts vs. Browsing Knowledge in Hypertext Systems," *Computer*, January 1988, p. 70.

McAleese, Ray. "Hypertext: Theory into Practice," edited by Ray McAleese, ABLEX Publishing Corporation, Norwood, NJ 07648.

Messinger, Eli, Shoens, Kurt, Thomas, John, Luniewski, Allen. "Rufus: The Information Sponge," Research Report RJ 8294 (75655), August 13, 1991, IBM Almaden Research Center.

Metzler and Haas. "The Constituent Object Parser: Syntactic Structure Matching for Information Retrieval," *Proceedings of the ACM SIGIR Conference*, 1989, ACM Press.

Nelson, T.H. *Literary Machines*, self published by Nelson, Nashville, Tenn., 1981. Did much to popularize hypertext; at the time of writing he has still not released a working product, though competitors such as hypercard have done so with notable success.

Mozer, Nfichael C. "Inductive Information Retrieval Using Parallel Distributed Computation," UCLA.

Pike, Rob and Weinberger, P.J. "The Hideous Name," AT&T Research Report.

Pike, Rob, Presotto, Dave, Thompson, Ken, Trickey, Howard, Winterbottom, Phil. "The Use of Name Spaces in Plan 9," available via ftp from att.com. Plan 9 is an operating system intended to be the successor to Unix, and greater integration of its name spaces is its primary focus.

Potter, Walter D. and Trueblood, Robert P. "Traditional, semantic, and hyper-semantic approaches to data modeling," *Computer*, v21, 1988, p. 53(11).

Rijsbergen, C. J. Van, *Information Retrieval*, 2nd. ed., Butterworth and Co. Ltd., 1979, printed in Great Britain by The Whitefriars Ltd., London and Tonbridge.

Salton, G. "Another Look At Automatic Text-Retrieval Systems," *Communications of the ACM*, v29, 1986, pp. 648-656.

Smith, J.M. and Smith, D.C. "Database Abstractions: Aggregation and Generalization," *ACM Transactions Database Systems*, June 1977, pp. 105-133, ICS Report No. 8406, June 1984.

Corbató, F. J. and Vyssotsky,V. A. "Introduction and Overview of the Multics System," this volume.

Couleur, J.F., Glaser, E.L., and Oliver, G. A. "System Design of a Computer for Time-Sharing Applications," this volume.

Corbató, F.J., Graham, R.M., and Vyssotsky, V. A."Structure of the Multics Supervisor," this volume.

Dunten, S. D., Mikus, L.E. and Ossana, J. F. "Communications and Input-Output Switching in a Multiplex Computing System," this volume.

David Jr., E. E., and Fano, R. M. "Some Thoughts About the Social Implications of Accessible Computing," this volume.

Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation For UNIX Development." In *Proceedings of the USENIX 1986 Summer Conference*, June 1986.

Bach, M. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

Bina, E. and Emrath, P. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Conference*, January 1989.

Card, R., Commelin, E., Dayras, S. and Mével, F. The MASIX Multi-Server Operating System. In *OSF Workshop on Microkernel Technology for Distributed Systems*, June 1993.

SECURITY INTERFACE for the Portable Operating System Interface for Computer Environments - Draft 13. Institute of Electrical and Electronics Engineers, Inc, 1992.

Kleiman, S. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer USENIX Conference*, pp. 260-269, June 1986.

Fabry, R., Joy, W., McKusick, M. and Leffler, S. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.

Bostic, K., McKusick, M., Seltzer, M. and Staelin, C. "An Implementation of a Log-Structured File System for UNIX." In *Proceedings of the USENIX Winter Conference*, January 1993.

Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice Hall, 1987.

Additional References

Bachman, C. W. and Williams, S. B. "A General Purpose Programming System for Random Access Memories," *Proceedings of the Fall Joint Computer Conference*, 26, Spartan Books, Baltimore, 1964.

Dennis, J. B. and Van horn, E. C. "Programming Semantics for Multiprogrammed Computations," *ACM Conference on Programming Languages*, San Dimas, Calif., Aug. 1965. To be published in *Comm. ACM*.

Holt, W. "Program Organization and Record Keeping for Dynamic Storage Allocation," *Comm. ACM* 4, pp. 422-431, Oct. 1961.

Nelson, T. H. "A File Structure for the Complex, the Changing and the Indeterminate," *ACM National Conference*, Aug. 1965.

Wilkes, M. V. "A Programmer's Utility Filing System," *Computer Journal 7*, pp. 180-184, Oct.1964.

부록 3. Loopback뿌리파일체계의 리용방법

이 리용방법은 구획을 재구성하지 않고 DOS구획으로부터 실행할수 있는 Linux고유파일체계방식의 설치판을 생성하는데 Linux loopback장치를 어떻게 리용하는가를 설명하고 있다. 이와 꼭 같은 기술들중에서 다른 사용방법들도 역시 토론된다.

1. 요 약

1.1. 저작권

Loopback Root Filesystem 참고서 Copyright © 1998-99 Andrew M.Bishop (amb@gedanken.demon.co.uk)

이 문서는 개방문서이다. 사용자는 무료쏘프트웨어재단에 의하여 출판된 GNU일반공개사용허가권의 협정하에 누구나 이것을 재배포할수 있으며 또한 변경시킬수 있다. 더 구체적으로 알기 위해서는 GNU GPL을 참조하면 된다.

1.2. 개정력사

판본 1.0.0 초기판본(1998.6)
판본 1.0.1-1.0.3 약간의 변경과 핵심부판본변경 OS형 등(1998-1999.6)
판본 1.1 저작권과 재허가권보충(1999.9)

2. Loopback장치와 Ramdisk의 원리

먼저 뿌리장치로서 loopback를 설정하는데 리용된 몇가지 일반적원리를 서술한다.

2.1. Loopback장치

Linux에서 Loopback장치는 어떤 다른 매체장치로 사용할수 있는 가상장치이다. 표준매체장치의 실례로는 /dev/hda1, /dev/hda2. /dev/sda1과 같은 하드디스크구획들이거나 혹은 플로피디스크 /dev/fdo 등과 같은 전체 디스크들을 들수 있다.

이것들은 모두 파일이나 등록부구조를 보존하는데 리용할수 있는 장치들이다. 이 장치들은 요구되는 파일체계(ext2fs, msdos, ntfs 등)로 양식화될수 있으며 그다음 올려태우기될수 있다. Loopback파일체계는 다른 파일체계의 파일과 완전한 장치로서 련결된다. 또한 이 체계는 우에서 목록으로 보여 준 임의의 다른 장치와 같이 양식화될수 있으며 올려태우기될수 있다.

이를 위하여 /dev/loop0 혹은 /dev/loop1이라고 부르는 장치가 파일과 련관되며 이 새로운 가상적장치가 올려태우기된다.

2.2. Ramdisk장치

Linux에서는 또한 파일체계로서 올려태우기된 다른 형의 가상장치들을 가질수 있는데 이것이 바로 Ramdisk장치이다.

이 경우에 장치는 어떤 물리적장치로 간주되지 않고 그 목적과는 별개로 설정한 기억의 한 부분이다. 배정된 기억은 디스크밖으로는 절대로 교환되지 않지만 디스크캐쉬에는 남아 있게 된다. ramdisk는 임의의 순간에 ramdisk장치 /dev/ram0 혹은 /dev/ram1에 써넣는 방법으로 생성할수 있다. 다음 이 디스크는 loopback장치에서와 같은 방법으로 양식화되고 올려태우기될수 있다. ramdisk가 기동하는데 리용될 때(보통 Linux설치디스크나 혹은 회복디스크) 디스크이메지가(단일파일로서 디스크의 전체 내용) 압축된 형식으로 기동플로피디스크상에 기억될수 있다.

이 조작은 기동시 핵심부에 의하여 자동적으로 인식되며 그것이 올려태우기되기전에 압축이 해제되여 ramdisk에 적재된다.

2.3. 초기 Ramdisk장치

Linux에서 초기 ramdisk장치는 뿌리파일체계로 loopback장치를 사용할수 있게 할 요구로부터 나온 하나의 중요한 기술이다.

초기 ramdisk는 파일체계이메지에 리용될 때 기억에 복사되고 올려태우기되며 따라서 거기에 있는 파일들에 접근이 가능하게 된다. 이 ramdisk우의 프로그람이(/linuxrc라고 부른다.) 실행되여 그것이 완료되면 다른 장치가 뿌리파일체계로서 올려태우기된다. 그렇지만 변경된 ramdisk는 여전히 존재하며 /initrd등록부에 올려태우기되거나 /dev/initrd장치를 거쳐 리용될수 있다. 표준기동순서는 지적된 뿌리구획으로부터 기동하며 실행을 계속한다. 초기 ramdisk선택항목을 리용하여 뿌리구획은 기본기동순서가 시작되기전에 변경가능하게 된다.

2.4. 뿌리파일체계

뿌리파일체계는 기동후에 /라는 등록부로 나타날수 있도록 제일 처음에 올려태우기되는 장치이다. 모든 파일들을 포함하고 있다는 사실로부터 그 파일들이 있는 뿌리파일체계에는 복잡성이 많다. rc스크립트에 대한 기동이 실행될 때 이 프로그람들은 /etc/rc.d에 있는 파일이거나 /etc/init프로그람의 판본에 따라 /etc/rc?.d에 있는 파일일수도 있다. 그 근거는 초기 ramdisk가 마지막뿌리구획이 기동시에 적재된 구획과 꼭 같아 지지 않도록 하는데 리용될수 있으므로 쓸모있다는데 있다.

2.5. Linux기동순서

초기 ramdisk가 기동시에 어떻게 동작하는가를 보기 위하여 사건의 순서를 아래에 보여 준다.

1. 핵심부가 기억에 적재되는데 이 조작은 LILO나 혹은 LOADING에 의하여 수행된

다. 이 과정의 실행은 Loading…통보문을 통해 알수 있다.

2. ramdisk이메지는 기억에 적재되며 이것은 다시 LILO 혹은 LOADING에 의하여 수행된다. 사용자는 이 과정을 역시 Loading…통보문을 통하여 볼수 있다.
3. 명령행선택항목의 문장론적해석과 뿌리장치로서 ramdisk의 설정을 포함하는 핵심부의 초기화가 진행된다.
4. 초기 ramdisk상에서 /linuxrc프로그람이 실행된다.
5. 뿌리장치가 핵심부파라메터에 정의된것으로 변경된다.
6. 사용자배치구성기동조작이 차례로 수행될 /etc/init프로그람이 실행된다.

3. Loopback뿌리장치의 생성방법

먼저 일반적원리를 설명하고 lookback장치를 생성하는 방법을 서술한다.

3.1. 요 구

loopback뿌리장치를 생성하기 위하여 여러가지 요구가 제기된다.

▼ 작업용Linux체계
▲ 목적DOS구획에 큰 파일을 복사하는 방법

가장 중요한것은 설치된 Linux체계에 대한 접근이다. 접근이 중요한것은 loop장치가 Linux하에서만 생성될수 있기때문이다. 이것은 아무것도 없는데서 동작하는 체계를 기동할수 없다는것을 의미한다. 사용할 Linux체계에 대한 요구는 사용자가 체계우에 핵심부를 콤파일해야 한다는것이다.

일단 loopback장치가 생성되면 생성된 파일은 한개의 큰 파일로 된다. 80MB파일을 사용했을 때 X말단을 리용하는데는 충분하지만 다른 프로그람을 더 사용하려고 하면 그리 충분하지 못하다. 이 파일은 DOS구획우에 복사되여야 하며 망이라든지 혹은 여러가지 플로피디스크들을 리용할수 있게 되여야 한다.

사용자에게 요구되는 프로그람들에는 다음과 같은것들이 포함될수 있다.

▼ LOADLIN판본 1.6 혹은 그이상
- loopback장치를 지원하는 올려태우기판본
- 요구되는 선택항목을 지원하는 핵심부판본

▲ 모든 프로그람들은 최근의 Linux설치프로그람에 대하여 표준으로 되여야 한다.

3.2. Linux핵심부생성

Linux판본 2.0.31을 리용하여 loopback2장치를 생성하였다. 다른 판본들도 동작할수 있지만 그것들은 적어도 다음과 같은 선택항목들을 가지고 있어야 한다.

리용하는데 필요한 핵심부선택항목들은 다음과 같다.

▼ RAM디스크지원(CONFIG_BLK_DEV_RAM)
- 초기 RAM디스크(initrd)지원(CONFIG_BLK_DEV_INITRD)
- Loop장치지원(CONFIG_BLK_DEV_LOOP_
- FAT파일체계지원(CONFIG_FAT_FS)

▲ MS-DOS파일체계지원(CONFIG_MSDOS_FS)

첫 두개 항목은 RAM디스크장치자체의 초기 RAM디스크에 대한것이다.

다음의것은 loopback파일체계항목이다. 마지막두개는 MSDOS파일체계지원을 위한것인데 이것은 DOS구획을 올려태우기하는데 필요하다.

사용자가 모듈을 요구하며 또 그것이 가능하다고 해도 제일 쉬운 방법은 모듈없이 핵심부를 콤파일하는것이다.

사용자가 가지고 있는 핵심부의 판본에 따라 핵심부검사수정을 적용해야 한다. 이것들은 loopback장치를 뿌리파일체계로 사용할수 있게 하는 아주 간단한것들이다.

▼ 2.0.0이전의 핵심부판본 ; 이것들에 대한 정보는 없다.
- 핵심부판본 2.0.0~2.0.34 ; 사용자는 아래에 보여 준것처럼 2.0.x핵심부에 대하여 핵심부검사수정을 적용하여야 한다.
- 핵심부판본 2.0.35~2.0.x ; 핵심부검사수정이 요구되지 않는다.
- 핵심부판본 2.1.x ; 2.0.x 혹은 2.2.x에 대하여 핵심부검사수정을 적용해야 한다.
- 핵심부판본 2.2.0~2.2.10 ; 아래에 보여 준것처럼 2.2.x핵심부에 관하여 핵심부검사수정을 적용해야 한다.

▲ 핵심부판본 2.3.x ; 아래에 보여 준것처럼 2.2.x핵심부에 핵심부검사수정을 적용해야 한다.

2.0.x핵심부에 대하여 /init/main.c는 아래의 변경된 판본에 보여 준것처럼 한개의 행을 보충할것을 요구한다.

```
static void parse_root_dev(char*line)
 {
   int base=0;
   static struct dev_name_struct{
          const char*name;
          const int num;
       }devices[]={
         { “ nfs”,   0x00ff},
         { “ loop”,  0x0700},
         { “ hda”,   0x0300},
         { “sonycd”, 0x1800},
         {NULL, 0}
       };
```

```
…
}
```

2.2.x핵심부에서 /init/main.c는 아래에 보여 준것처럼 3개 행을 보충할것을 요구한다. “loop”, 0x0700과 그열에 있는것들을 포함하는 행이 보충된것들이다.

```
static struct dev_name_struct{
       const char *name;
       const int num;
  }root_dev_names[] __initdata={
  #ifdef CONFIG_ROOT_NFS
      {“ nfs”,0x00ff},
  #endif
  #ifdef CONFIG_BLK_DEV_LOOP
      {“ loop”,    0x0700},
  #endif
  #ifdef CONFIG_BLK_DEV_IDE
      {“ hda”,     0x0300},
…
        {“ ddv”,DDV_MAJOR << 8},
  #endif
      {NULL, 0}
};
```

일단 핵심부가 배치구성을 완료하면 zImage파일을 생성하기 위하여 콤파일하여야 한다. 이 파일은 콤파일될 때 arch/i386/boot/zImage에 존재하게 된다.

3.3. 초기Ramdisk장치생성

초기ram디스크는 출발로부터 시작하여 loopback장치로 아주 쉽게 생성된다. 사용자는 이 조작을 뿌리에 관하여 진행하게 된다. 사용자가 실행해야 할 명령들을 아래에 보여 준다. 이 명령들은 뿌리의 홈등록부(/root)로부터 실행되는것으로 가정한다.

```
mkdir /root/initrd
dd if=/dev/zero of=initrd.img bs=1k count=1024
mke2fs -i 1024 -b 1024 –m 5 -F -v initrd.img
mount initrd.img/root/initrd -t ext2 -o loop
 cd initrd
[create the files]
```

```
cd..
umount /root/initrd
gzip -c -9 initrd.img>initrdgz.img
```

이것을 실행하기 위한 단계들을 아래에 제시한다.

1. 초기ram디스크를 위한 올려태우기위치를 생성한다(빈등록부로서).
2. 요구되는 크기의 빈파일을 생성한다. 여기서는 1024KB를 사용하였는데 내용에 따라 많이 혹은 적게 요구할수도 있다(크기는 마지막파라메터).
3. 빈파일에 ext2파일체계를 만든다.
4. 파일을 올려태우기위치에 올려태우기한다. 이 조작은 loopback장치를 리용한다.
5. 올려태우기된 loopback장치로 변경한다.
6. 요구되는 파일들을 생성한다(자세한것은 아래부분을 참고).
7. 올려태우기된 loopback장치를 이전시킨다.
8. 장치의 올려태우기를 해제한다.
9. 후에 리용하기 위하여 압축판본을 생성한다.

초기ram디스크의 내용

사용자의 ram디스크에서 필요되는 파일들은 임의의 명령들을 실행시킬수 있는 최소요구에 해당한다.

/linuxrc msdos파일체계를 올려태우기하기 위하여 실행되는 스크립트(아래를 보라.)
/lib/* 동적프로그람이 요구하는 련결편집기와 서고들
/etc/* 동적련결편집기에 의해 리용되는 캐쉬
/bin/* 쉘해석기(bash보다 더 작으므로 ash)

DOS디스크들의 조종과 loopback장치들의 설치에 필요한 올려태우기 및 적재설치프로그람들

/dev/* 사용할 장치들. Ldlinux.so에 대하여 /dev/zero가 필요하다.
/dev/hda* MSDOS디스크와 loopback장치를 위한 /dev/loop*을 올려태우기하기 위하여 필요하다.
/mnt MSDOS디스크를 올려태우기하기 위한 자유목록

사용한 초기 ram디스크들을 아래에 보여 준다. 내용은 파일체계의 휴지시간을 고려할 때 약 800KB정도 된다.

```
total 18
drwxr-xr-x   2 root   root    1024 Jun  2 13:57 bin
drwxr-xr-x   2 root   root    1024 Jun  2 13:47 dev
drwxr-xr-x   2 root   root    1024 May 20 17:43 etc
drwxr-xr-x   2 root   root    1024 May 27 07:57 lib
```

```
drwxr-xr-x   1 root   root     964 Jun  3 08:47 linux rs
drwxr-xr-x   2 root   root   12288 May 27 08:08 lost+found
drwxr-xr-x   2 root   root    1024 Jun  2 14:16 mnt

./bin:
total 168
-rwxr-xr-x   1 root   root   60880 May 27 07:56 ash
-rwxr-xr-x   1 root   root    5484 May 27 07:56 losetup
-rwxr-xr-x   1 root   root   28216 May 27 07:56 mount
lrwxrwxrwx   1 root   root       3 May 27 08:08 sh->ash
./dev:
total 0
brw-r—-r--  1 root   root   3,   0 May 20 07:43 hda
brw-r-—r--  1 root   root   3,   1 May 20 07:43 hda1
brw-r—-r--  1 root   root   3,   2 Jun  2 13:46 hda2
brw-r—-r--  1 root   root   3,   3 Jun  2 13:46 hda3
brw-r—-r--  1 root   root   7,   0 May 20 07:43 loop0
brw-r—-r--  1 root   root   7,   1 Jun  2 13:47 loop1
crw-r—-r--  1 root   root   1,   3 May 20 07:42 null
crw-r—-r--  1 root   root   5,   0 May 20 07:43 tty
crw-r—-r--  1 root   root   4,   1 May 20 07:43 tty1
crw-r—-r--  1 root   root   1,   5 May 20 07:42 zero

./etc:
total 3
-rw-r—-r--  1 root root      2539 May 20 07:43 1d.so.cache

./lib:
total 649
lrwxrwxrwx   1 root root       18 May 27 08:08 ld_linux.so.1->
                                 ld-linux.so.1.7.14
-rwxr-xr-x   1 root root    21367 May 20 07:44 ld-linux.so.1.7.14
lrwxrwxrwx  1 root toor       14 May 27 08:08 lib.so.5->libc.so.5.3.12
-rwxr-xr-x   1 root root   583795 May 20 07:44 libc.so.5.3.12

./lost+found:
 total 0

./mnt:
```

```
total 0
```

우의것들에 대하여 유일하게 복잡한 단계는 dev에 있는 장치들에 있다. 장치들을 생성하기 위하여 mknod를 사용하며 요구되는 파라메터들을 얻기 위한 형타적인 요소로 /dev안에 있는 현존장치들을 리용한다.

/linuxrc파일

초기ram디스크의 /linuxrc파일은 loopback장치들이 뿌리구획에 리용될수 있도록 모든 준비를 갖추어 주는데 리용된다.

다음의 실례는 /dev/hda1을 MSDOS구획으로 올려태우기하며 만일 올려태우기가 성공하면 /linux/linuxswp.img파일을 /dev/loop0으로 설치하며 /linux/linuxswp.img를 /dev/loop1로 설정하는 경우의 실례이다.

```
#!/bin/sh
echo INITRD:Trying to mount/dev/hda1 as msdos
if/bin/mount –n -t msdos /dev/hda1 /mnt; then
    echo INITRD:Mounted OK
  /bin/losetup/dev/loop0/mnt/linux/linuxdsk.img
  /bin/losetup/dev/loop1/mnt/linux/linuxswp.img
  exit 0
else
 echo INITRD;Mount faild
 exit 1
fi
```

첫번째 장치 /dev/loop0은 뿌리장치로 되며 두번째 장치 dev/loop1은 교환용예비장치로 된다.

만일 사용자가 DOS구획을 뿌리 아닌 사용자로서 쓸수 있게 하려면 mount −n −t msdos/dev/hda1/mnt−0 uid=0, gid=0, umask=000을 리용해야 한다. 이 조작은 DOS구획에 대한 접근을 뿌리로 넘기게 되며 적합한 허가권을 설정한다.

3.4. 뿌리장치생성

사용자가 리용하게 될 뿌리장치는 linuxdsk.img파일이다. 사용자는 초기ram디스크를 생성할 때와 꼭 같은 방법으로 이 뿌리장치를 생성하여야 하는데 더 크게 생성할 필요는 없다.

이 디스크에 사용자가 좋아 하는 임의의 linux설치프로그람을 설치할수 있다.

제일 쉬운 방법은 현존 linux설치프로그람을 디스크로 복사하는것이다. 다른 방법은 새로운 linux설치프로그람을 디스크에 설치하는것이다. 이 조작이 수행되였다고 가정하면 일부 약간의 변경을 가해 주어야 한다.

/etc/fstab파일은 뿌리구획과 초기 ram디스크에 설정되는 두개의 loopback장치를 리용하는 교환조작을 참조하여야 한다.

```
/dev/loop0  /    etc2 defaults 1 1
/dev/loop1 swap  swap defaults 1 1
```

이 조작은 실지 뿌리장치가 리용될 때 핵심부가 뿌리장치가 있는 곳을 혼돈하지 않는다는것을 담보한다. 또한 이 조작은 교환구획이 표준적으로 리용될 때와 같은 방법으로 교환공간을 첨가할수 있게 한다. 사용자는 뿌리디스크장치나 혹은 교환구획에 대한 그 어떤 다른 참조값을 제거해야 한다.

만일 Linux가 출발한 다음에 DOS구획을 읽을수 있게 하려면 일정한 여유를 가지고 약간의 변환을 주어야 한다.

▼ /initrd라는 등록부를 만든다. 이 등록부는 일단 loopback뿌리파일체계가 올려태우기되면 초기ram디스크가 올려태우기되는 장소이다.
- 실지 DOS구획이 올려태우기될 /initrd/mnt를 지적하는 /DOS라는 기호적련결을 생성한다.

▲ 디스크를 올려태우기하는 rc파일에 한개 행을 보충한다. 이 조작은 mount−f−t msdos/dev/hda1/initrd/mnt명령을 실행해야 한다. 이 조작은 DOS구획에 "fake"올려태우기를 생성하게 되며 따라서 모든 프로그람들은(df와 같은) DOS구획에 올려태우기되며 그것을 어디서 찾을수 있는가를 알수 있게 된다. 만일 사용자가 /linuxrc파일에서 다른 선택항목을 리용했다면 명백히 그 항목을 여기서 리용해야 한다. 초기에 이미 적재된 다음부터는 이 뿌리장치에 Linux핵심부를 설정할 필요가 없다. 그러나 만일 모듈들을 리용한다면 그 모듈들을 표준적으로 이 장치에 포함시켜야 한다.

3.5. 교환장치생성

사용자가 리용할 뿌리장치는 linuxswap.img파일이다. 교환장치는 만들기가 아주 쉽다. 초기ram디스크를 만들 때처럼 빈파일을 만들고 그것을 초기화하기 위하여 mkswap linuxswap.img를 실행한다.

교환공간의 크기는 사용자가 설치된 체계로 무슨 작업을 하는가 하는데 의존하는데 사용자가 가지고 있는 RAM의 크기와 8MB사이에서 선택할것을 권고한다.

3.6. MSDOS등록부생성

사용하려고 하는 파일들을 DOS구획으로 옮겨야 할 필요가 있다. C : \Linux라고 부르는 DOS등록부에서 요구되는 파일들은 다음과 같은것들이다.

▼ LinuxDSK. IMG뿌리장치로 되는 디스크이메지

▲ LinuxSWP. IMG교환공간

3.7. 기동플로피디스크생성

여기에서 리용할 기동디스크는 곧 표준DOS양식의 기동가능한 플로피디스크를 말한다. 이 디스크는 DOS상에서 format a : /s명령을 리용하여 생성할수 있다. 이 디스크상에

서 사용자는 AUTOEXEC. BAT파일(아래에서와 같이)을 만들어야 하며 핵심부를 복사하고 실행가능한 LOADLIN과 압축초기 ram디스크를 생성해야 한다.

▼ AUTOEXEC. BAT DOS를 자동적으로 실행하는 일괄파일
- LOADLIN. EXE 실행가능한 LOADLIN프로그람
- ZIMAGE Linux핵심부

▲ INLTRDGZ.IMG 압축된 초기ram디스크

AUTOEXEC.BAT파일은 아래와 같은 한개의 행을 포함하여야 한다.

```
\loadlin/zImage initrd=\initrdgz. img root=/dev/loop0 ro
```

이 명령은 사용할 핵심부이메지, 초기ram디스크이메지 그리고 초기ram디스크가 완료된후의 뿌리장치들을 정의하며 뿌리구획이 읽기방식으로만 올려태우기될것이라는것을 정의한다.

4. 체계의 기동

새 기동장치로부터 체계를 기동하기 위하여 요구되는것은 우에서 설명한 과정을 통하여 준비된 플로피디스크를 개인용콤퓨터에 끼워 넣는것이다.

기동을 위한 동작순서는 다음과 같다.

1. DOS기동
2. AUTOEXEC. BAT출발
3. LOADLIN실행
4. Linux핵심부의 기억에로의 복사
5. 초기 ram디스크의 기억에로의 복사
6. Linux핵심부의 실행시작
7. 초기ram디스크상의 /linuxrc파일실행
8. DOS구획의 올려태우기, 뿌리장치와 교환장치설정
9. loopback장치로부터 기동순서렬의 계속 실행

이 동작들이 완성된후 기동디스크를 해제하고 Linux체계를 사용한다.

4.1. 해결가능한 문제점

우의 공정들에는 실패할수 있는 여러 단계들이 존재하는데 그것들이 어떤것들이며 또 무엇을 검사하겠는가를 설명하려고 한다. DOS기동과정은 화면상에 표시되는 MS-DOS starting…이라는 통보문을 보고 쉽게 알수 있다. 만일 이 통보문이 보이지 않으면 플로피디스크가 기동할수 없는 상태이거나 혹은 콤퓨터가 플로피디스크구동기에 의하여 기동할수 없다는것을 말한다.

AUTOEXEC.BAT파일이 파일안에 있는 명령들을 수행할 때 기정값설정에 의하여 화면상에 표시되지 않는것이다. 이 경우에는 LOADLIN을 시작하는 파일안에 한개의 명령

행이 있게 된다.

LOADLIN을 실행할 때 이 명령행은 두가지 눈에 뜨이는 동작을 수행한다. 하나는 핵심부를 기억에 적재하는것이고 다른 하나는 ram디스크를 기억에 복사하는것이다. 이 조작은 둘다 Loading… 통보문에 의하여 지시된다.

핵심부는 그자체가 압축되지 않은 상태에서 출발한다. 이 조작은 핵심부의 이메지가 손상될 때 crc오유들을 발생시킬수 있다. 그다음 핵심부는 직렬화순서렬을 실행하기 시작한다. 이때 직렬화순서렬은 장황한 진단통보문을 내보내게 된다. 초기ram디스크장치의 적재과정 역시 이 단계를 실행할 때 볼수 있다.

/linuxrc파일이 실행될 때는 진단통보문이 전혀 나타나지 않지만 오유수정도구의 지원으로 자체로 추가해 줄수 있다. 만일 이 단계에서 loopback장치들을 뿌리장치로 설치하는데 실패하면 뿌리장치가 없다는 통보문과 핵심부의 설치가 보류되였다는 통보문을 볼수 있을것이다.

새로운 뿌리장치에 대한 표준기동순서렬은 계속 진행되며 여기에는 장황한 내용들이 아주 많다. 읽기-쓰기방식으로 올려태우기되고 있는 뿌리장치에 대하여서는 문제점들이 있을수 있지만 LOADLIN명령행의 선택항목 "ro"을 리용하여 그것을 정지시킬수 있다. 다른 문제들도 발생할수 있는데 여기에는 기동순서렬이 뿌리장치가 있는 위치를 혼돈하여 생기는 경우가 속한다. 이 경우는 대체로 /etc/fstab와 관련되는 문제이다. 기동순서가 완료되면 나머지문제는 프로그람들이 DOS구획이 올려태우기되였는지 혹은 올려태우기되지 않았는지에 대하여 혼돈하는 문제이다. 이때는 이미 설명한 fake올려태우기명령을 사용하는것이 좋은 착상으로 될것이다. 이 명령은 DOS장치상에서 파일에 접근하려고 할 때 아무런 문제도 생기지 않게 한다.

4.2. 도서목록

첫 loopback뿌리파일체계를 만들 때 참조한 문헌들을 아래에 소개한다.

▼ Linux핵심부원천 특히 init/main.c
- Linux핵심부문헌 특히 Documentation/initrd. txt와 Documentation/ramdisk.txt
- LILO문헌

▲ LOADLIN문헌

5. 다른 Loopback뿌리장치의 리용가능성

일단 DOS구획상의 파일체계에 의한 기동원리가 확립되기만 하면 사용자가 그것을 리용하여 다른 많은 문제들을 해볼수 있다.

5.1. DOS하드디스크설정

만일 기동플로피디스크에 의하여 DOS하드디스크상의 파일로부터 Linux를 기동할수 있으면 그것은 명백하게 하드디스크자체를 리용하여 역시 기동할수 있다는것을 말한다. 배치구성기동차림표로 AUTOEXEC. BAT안으로부터 LOADLIN실행을 위한 선택항목을

주는데 리용할수 있다. 이것을 리용하면 기동순서를 더 빨리 실행할수 있는데 그렇지 않은 경우에는 같다.

5.2. LILO기동설치

LOADLIN의 리용은 Linux핵심부의 기동을 위한 한개 선택항목에 불과하다. LILO도 역시 꼭 같은 동작을 수행하지만 DOS를 필요로 하지 않는다. 이 경우에 DOS양식플로피디스크는 ext2양식의 디스크로 교체될수 있다. 그밖의 구체적내용들은 핵심부와 디스크에 자료를 가지고 있는 초기ram디스크와 아주 류사하다.

LOADLIN방법을 선택하게 된것은 LILO에 주어 져야 할 인수들이 다소 복잡하다는데 있다. 역시 LOADLIN은 DOS조종하에서 읽을수 있기때문에 유연디스크가 무엇인가를 무심히 알게 되는 사람들에게는 더 명백하다.

5.3. VFAT/NTFS설치

NTFS방법에서는 아무러한 문제도 없었다. NTFS파일체계구동기는 판본 2.0에서 표준핵심부선택항목은 아니지만 http : //www. infomatik. hu−berlin. de/~loewis/ntfs/로부터 검사수정으로 리용할수 있다. 판본 2.2.x에서는 NTFS구동기가 핵심부안에 표준으로 포함된다.

VFAT 혹은 NTFS선택항목들에서는 유일한 변화들이 초기ram디스크에 있다. /linuxrc파일에 대해서는 MS−DOS보다는 VFAT나 NTFS형파일체계를 올려태우기할 필요가 있다. 이 방법이 VFAT구획상에서도 작업할수 있기때문이다.

5.4. 재구획분할이 없이 Linux설치

표준배포판으로부터 개인용콤퓨터상에 Linux를 설치하는 공정은 플로피디스크로부터의 기동과 디스크의 구획재분할을 요구한다. 이 단계는 빈 loopback장치와 교환파일을 생성하는 유연성기동디스크에 의하여 이루어 진다. 이 과정은 설치가 표준적으로 진행될수 있게 하지만 구획보다는 오히려 loopback장치에 설치되게 한다.

이 단계는 UMSDOS설치와 다른 설치에도 리용될수 있다. ext2파일체계에서 최소배정단위가 DOS구획의 32KB대신에 1KB이기때문에 디스크의 리용상견지에서 아주 효과적이다. 또한 다른 방법으로는 문제가 생기는 VFAT나 NTFS양식화디스크들에서도 리용할수 있다.

5.5. 비기동장치로부터의 기동조작

이 방법은 표준적으로 기동할수 없는 장치로부터 Linux체계를 기동하는데도 역시 사용할수 있다.

▼ CD−ROM
- 압축디스크

▲ 병렬포구디스크구동기

다른 장치들에서도 많이 사용할수 있는데 NFS뿌리파일체계들은 이미 선택항목으로 핵심부에 포함되여 있지만 여기에서 서술한 방법도 역시 대신 리용할수 있다.

부록 4. Linux의 구획설정방법

이 Linux속성참고서는 사용자의 Linux체계에 디스크공간을 어떻게 계획하며 구성배치를 어떻게 하는가에 대하여 서술한다. 이 문서는 디스크장치, 구획, 교환공간크기설정, 위치적문제, 파일체계, 형식 그리고 련관된 사항들에 대하여 취급한다. 기본취지는 절차가 아니라 몇가지 기초적지식을 인식시키자는데 있다.

1. 요 약

1.1. 문서의 목적

이 문서는 Linux 2부류 참고서이다. 2부류 참고서는 Linux의 설치와 개별지도형식의 주장과 관련되는 일부 실무적문제를 해설해 주는 간단한 문서이다. 이 문서가 2부류의 소규모적문서로 되는 리유는 취급하는 내용이 너무 작아서 실지적인 참고서와 책이라기보다 본문이나 주제정도의 사용설명서에 불과하다.

1.2. 문서의 기본내용과 련관문서

이 특정한 2부류 참고서는 사용자가 Linux체계를 위한 디스크공간을 어떻게 계획하며 어떻게 배치구성할것인가를 서술해 준다. 이 문서는 디스크장치, 구획, 교환공간크기설정, 위치적문제, 파일체계, 파일체계형식과 관련 있는 주제들에 대하여 설명해 준다.

기본목적은 몇가지 기초적지식을 주는데 있으며 따라서 도구가 아니라 편리를 기본으로 하였다. 편의상 이 문서는 설치하기전에 읽어야 하는데 어쨌든 대부분의 사람들에게는 어렵다. 처음에는 디스크구조적배치최량화보다 다른 문제들이 제기되였다. 그리하여 사용자들속에서 Linux설치를 완료한 다음에 설치를 최량화하기 위한 방법과 계산정확도를 높이기 위한 방법을 생각하기 시작하였다. 이 본문을 읽고 Linux설치를 완료체계로 다시 해체하고 재구성해 보고 싶은 욕망이 생기게 될것이다. 이 2부류 참고서는 자체가 디스크공간을 대역분할하고 계획하는데서 일정한 제한을 가지고 있다. 여기서는 fdisk의 리용과 LILO, mke2fs나 혹은 여벌복사프로그람은 론의하지 않는다. 이러한 문제들을 취급하는 다른 참고서들이 있다. LinuxHOWTO에 관한 현행문헌들에 대하여서는 LinuxHOWTO색인을 보면 된다. 색인으로 HOWTO문서들을 얻을수 있는 지도서들도 나와 있다.

- ▼ 파일체계의 서로 다른 부분들에 대한 여러가지 크기와 속도요구를 어떻게 평가하는가를 알기 위하여서는 Gjoen Stein "Linux Multiple Disks Layout mini-HOWTO"를 보면 된다.
- 1024이상의 실린더를 가진 디스크를 고찰하는 지도서와 고려사항은 Andries Brouwer가 쓴 "Linux Large Disk mini-HOWTO"를 보면 된다.

▲ 사용자당(배정량) 디스크공간리용의 제한에 대한 지도서로는 Albert M. C. Tam이 쓴 "Linux Quota mini-HOWTO"를 보면 된다.

현재 디스크의 여벌복사에 대한 일반적인 문헌은 없지만 특정한 여벌복사해결안에 대한 지적자를 가지는 몇가지 문헌들은 있다. Linux를 IBM ADSM여벌복사환경으로 종합하는 지도서로는 Thomas Koening이 쓴 "Linux ADSM Backup mini-HOWTO"를 보면 된다. MS-DOS구동형Linux여벌복사에 대한 정보는 Christopher Neufeld가 쓴 "Linux Backup with MS-DOS mini-HOWTO"를 볼수 있다.

HOWTO문헌의 제시와 기록에 대한 지도서로는 Tim Bynum이 쓴 "Linux HOWTO Index"를 보며 /usr/src/Linux/Documentation을 브라우저를 통하여 열람하는 방법으로도 역시 도움을 받을수 있다. 사용자의 디스크구동기의 속성에 관한 몇가지 기본정보는 ide.txt와 scsi.txt를 참고할수 있으며 파일체계/부분등록부를 찾아 볼수 있다.

2. 구획에 대한 개념

개인용콤퓨터의 하드디스크들은 체계가 비록 한개의 디스크만을 가지고 있었지만 다중조작체계를 설치해 보려는 사람들에 의하여 즉시 발명되였다.

그로부터 단일한 물리적디스크를 다중론리디스크들로 분할하기 위한 기술이 요구되였다. 바로 그것이 구획이다. 완전히 분리된 디스크와 같이 취급되는 사용자하드디스크의 련속적인 블로크부분은 거의나 조작체계에 의하여 실현되였다. 구획들이 중복되지 말아야 한다는것은 명백하나 같은 기계에 설치된 서로 다른 조작체계가 중복구획들때문에 중요한 정보들이 덧씌여 지면 안된다. 린접한 구획들사이에는 역시 틈이 없어야 한다. 만일 틈을 주어도 해롭지 않다면 구획들사이에 공간을 남겨 두는 방법으로 선행디스크공간을 랑비하게 된다. 디스크는 구획으로 완전히 분할될것을 요구하지 않는다. 사용자는 자기의 디스크끝에 일정한 공간을 남겨 둘것을 예견할수도 있다. 이때 사용자디스크의 끝은 여전히 설치된 조작체계의 그 어떤 부분도 지적하지 못한다. 설치가 사용자에 의하여 대부분 시간동안 리용되였다는것이 명백할 때 사용자는 이 나머지부분을 공간상에 구획으로 설정할수 있으며 거기에 파일체계를 배치한다. 구획들은 옮겨 질수 없으며 그안에 포함되여 있는 파일체계를 소거하지 않고는 재배치할수도 없다. 그런데로부터 구획의 재구성은 보통 여벌복사와 구획재구성시에 변경된 모든 파일체계를 회복하는 동작이 동반된다. 사실 구획재구성시에는 모든것이 뒤죽박죽된다는것이 명백한 공통적현상이며 따라서 사용자는 지어 fdisk들과 같은 프로그람으로 다쳐 놓기전에 특정한 콤퓨터의 임의의 디스크에 있는 정보들을 여벌복사해 두어야 한다.

디스크우에 어떤 파일체계형을 가진 일부 구획들은 실제적으로 자료를 잃어 버림이 없이 두개로 분할될수 있다. 실례로 MS-DOS를 재설치하지 않고 Linux설치를 위한 대역을 확보하기 위하여 MS-DOS구획을 두개로 분할하기 위한 "fips"라는 프로그람이 있다. 그러나 사용자들은 아직도 여전히 자기의 콤퓨터상의 모든것을 주의 깊게 여벌복사해 두지 않고서는 이 조작을 하려고 하지 않는다.

2.1. 여벌복사의 중요성

테프장치들은 여벌복사의 중요한 수단이다. 테프장치들은 속도도 빠르고 현실성이 있으며 사용하기 쉽다. 그리하여 사용자들은 흔히 혼란이 없이 완전히 자동적으로 여벌복사를 실현할수 있다.

여기서는 디스크조종장치로 구동되는 ftape를 념두에 두는것이 아니라 실제테프에 대하여 말한다. SCSI를 도입하는 경우를 고찰하자. Linux는 SCSI를 선천적으로 지원하지 못한다. 사용자는 ASPI구동기를 적재할것을 요구하지 못하며 Linux상태에서 정확한 HMA를 적재하지 못하며 또 일단 SCSI주적응기가 설치되면 보충적인 디스크, 테프장치 그리고 CD-ROM을 그우에 추가할수 있다. 그이상의 I/O주소, IRQ조작이나 마스트/슬라브 그리고 PIO준위정합은 없다. 고유한 SCSI주적응기들은 CPU를 더 적재하지 않고도 높은 I/O성능을 제공한다. 둔한 디스크동작하에서도 사용자는 괜찮은 응답시간을 경험적으로 얻을수 있다. 만일 사용자가 기본 USENET 새 소식원천으로 Linux체계를 리용하려고 계획하거나 혹은 ISP업무에 리용하려고 한다면 SCSI없이 체계를 전개하는 문제에 대해서는 생각조차 할수 없다.

2.2. 장치번호와 장치이름

인텔에 기초한 체계에서 구획의 수는 애당초 제한되였다. 초기의 구획표는 기동쎅터부분으로서 설치되였으며 오직 4개의 구획입구점공간만을 가지였다. 이 구획들은 지금 1차구획이라고 부른다. 사람들이 체계상에 보다 많은 구획을 가질것을 요구한다는것이 명백해 졌을 때 론리구획이 발명되였다.

론리구획의 수는 제한이 없다. 매개 론리구획은 다음의 론리구획에 대한 지적자를 포함하며 따라서 사용자는 잠재적으로 제한이 없는 구획입구점들을 가질수 있게 되였다. 호환성으로 하여 모든 론리구획이 차지한 공간은 계산되여야 하였다. 만일 사용자가 론리구획을 리용하고 있다면 하나의 1차구획의 입구점은 《확장구획》(extended partition)으로 표기되며 그것의 시작과 끝블로크는 사용자의 론리구획이 차지한 대역을 표시한다. 이것은 전체 론리구획에 지정된 공간이 련속적이여야 한다는것을 의미한다. 오직 한개의 확장구획만 있을수도 있다. fdisk프로그람은 하나이상의 확장구획보다 더 큰 구획을 생성할수 없다.

Linux는 구동기당 제한된 수의 구획보다 큰 구획을 조종할수 없다. 그리하여 Linux에서는 사용자가 4개의 1차구획(론리구획을 리용하면 그들중 3개는 리용할수 없다.)과 하나의 SCSI디스크상에 기껏해서 15개의 구획을 가질수 있다(IDE디스크상에서는 전체적으로 63개).

Linux에서 구획들은 장치파일에 의하여 표현된다. 장치파일은 형 c(캐쉬를 리용하지 않는 장치로서 《문자》장치) 혹은 형 b(캐쉬를 거쳐 리용되는 장치로서 《블로크》장치)의 파일이다. Linux에서 모든 디스크들은 블로크장치로만 표시된다. 다른 범용장치들과 달리 Linux는 《행》속성판본의 디스크와 그것의 구획을 제공하지 못한다.

장치파일의 유일하게 중요한 내용은 그것의 기본(major) 및 부(minor)장치번호인데

주로 파일크기대신 주어 진다.

```
$ ls  -l /dev/hda
brw-rw----     1  root  disk  3,  0  Jul  18  1994  /dev/hda
                                     ^    ^
                                     |    부장치번호
                                    주장치번호
```

장치파일에 접근할 때 주번호는 어느 장치구동기가 입출력조작을 수행하기 위하여 호출되고 있는가를 선택한다. 이 호출은 파라메터로서 부번호에 의하여 수행되며 이것은 구동기가 부번호를 어떻게 해석하는가에 전적으로 달려 있다.

구동기문서는 보통 구동기가 부번호를 어떻게 시동하는가를 서술해 준다. IDE디스크에 대한 문서는 /usr/src/Documentation/ide.txt에 있다. SCSI디스크들에 대하여서는 /usr/src/linux/Documentation/scsi.txt에서 기대할수 있겠지만 거기에는 없다. 믿을수 있는 구동기원천은 /usr/src/linux/driver/scsi/sd. c : 184−196에서 찾아야 한다. Peter Anvin의 구동기번호와 이름목록은 /usr/src/linux/Documentation/devices.txt에 있다. 여기서 전체 블로크장치 즉 주번호 3, 22, 33, 34(IDE용)와 SCSI디스크들을 위한 주번호 8을 찾을수 있다. 주번호와 부번호들은 다 한바이트로 되여 있는데 그 리유는 매 디스크당 구획수가 제한되여 있기때문이다.

관례에 따라 장치파일들은 어떤 이름을 가지고 있으며 많은 체계프로그람들은 콤파일되는 이 이름들에 대한 지식을 가지고 있다. 그러한 이름들로는 IDE디스크에 대하여 /dev/hd*, SCSI디스크들에 대하여서는 /dev/sd*로 표현하며 개별적디스크들은 a, b, c 등으로 번호를 붙인다. 그러므로 /dev/hda는 IDE디스크들중의 첫번째 디스크를 가리키며 /dev/sda는 SCSI디스크의 첫번째를 가리킨다. 두 장치들은 다 블로크디스크에서 출발하여 전체 디스크를 표현한다. 불량한 도구들을 가지고 이러한 장치들에 대하여 쓰기를 진행하면 디스크상의 기본기동적재프로그람과 구획표를 파괴하게 되며 따라서 디스크의 모든 자료를 못쓰게 만들거나 혹은 체계가 기동할수 없게 한다. 이런 현상이 나타나기전에 상태를 알아야 하며 여벌복사를 진행해야 한다.

디스크상의 1차구획은 1, 2, 3, 4이다. 따라서 /dev/hda1은 첫 IDE디스크의 첫번째 1차구획으로 된다. /dev/hda2는 두번째 2차구획 등으로 된다. 론리구획은 5이상의 번호를 가지는데 /dev/sdb5는 2차 SCSI디스크의 첫번째 론리구획으로 된다.

매개 구획의 입구는 그것을 지적하는 시작블로크와 끝블로크의 주소를 가지고 있다. 형은 어떤 형의 조작체계에 대한 특정한 구획을 가리키는 수자코드(한바이트)로 되여 있다. 호환성을 보장하기 위하여 구획형코드는 단계적으로 유일하지 않으며 따라서 두개의 조작체계가 같은 형코드를 리용할 가능성은 언제나 있다.

Linux는 교환구획에 대하여 Ox82, 고유("native")파일체계(대체로 ext2fs)에 대하여서는 Ox83의 형코드를 보존하고 있다. 한때 대중화되였고 지금은 구식에 불과한 Linux/Minix파일체계는 구획형코드로 Ox81을 리용하였다. OS/2는 구획형코드를 Ox07로 표기하며 WindowsNT의 NTFS도 이 값을 리용한다. MS−DOS는 FAT파일체계의 특색에

맞게 여러가지 형코드를 배정한다. 그러한 형코드로는 Ox01, Ox04, Ox06이 알려 져 있다. DR-DOS는 어떤 때는 Linux/Minix와 충돌하기도 하면서 보호방식용구획을 지적하는데 Ox81을 리용하였지만 Linux/Minix나 DR-DOS에 더이상 리용하지 않는다. 론리구획의 저장통으로 리용되는 확장구획은 Ox05형코드를 리용한다.

구획은 fdisk프로그람으로 생성하거나 소거한다. 매개 자체조작체계프로그람은 fdisk를 리용하며 전통적으로 거의 모든 조작체계들에서 fdisk(혹은 fdisk.exe)라고 부르고 있다. 일부 fdisk들은 다른 조작체계들의 구획을 취급할 때 제한성을 가지게 된다. 이러한 제한성으로 하여 다른 체계의 형코드로 객체를 취급할수 없으며 1024개이상의 실린더를 다룰수 없고 구획의 생성이나 지어 실린더경계에 구획의 끝이 놓이지 않는다는것도 알수 없게 된다. 실례로 MS-DOS의 fdisk는 NTFS의 구획을 지울수 없으며 OS/2의 fdisk는 Linux의 fdisk에 의하여 생성된 실린더경계에 끝이 놓이지 않는 구획을 《정확한》것으로 리해하며 DOS와 OS/2은 둘다 1024개이상의 실린더를 가진 디스크에서 비정상상태를 나타낸다(이 디스크들에 대한 자세한 내용은 “large-disk”Mini-Howto를 참조할것).

3. 구획의 기본내용

3.1. 구획의 분할

일부 조작체계들은 정상적인 사고를 초월하는 리유들로 하여 론리구획으로부터의 기동을 믿지 않는다. 대체로 사용자들은 MS-DOS나 OS/2 그리고 Linux가 리용할 조작체계를 위한 기동구획으로서 1차구획을 예약하려고 한다. 하나의 구획이 확장구획으로 되여야 하며 그 확장구획은 론리구획을 포함하는 디스크의 나머지부분들에 대하여 저장통으로 쓰인다는것을 잊지 말아야 한다.

조작체계의 기동은 BIOS와 1024개실린더한계를 포함하는 실방식기동이다. 그러므로 사용자는 이러저러한 문제들을 피하기 위하여 자기의 모든 기동구획을 하드디스크의 첫 1024개실린더에 놓으려고 한다.

Linux를 설치하기 위하여서는 사용자에게 적어도 한개의 구획이 있어야 한다. 만일 핵심부가 이 구획으로부터 적재된다면(실례로 LILO에 의하여) 이 구획은 반드시 BIOS에 의하여 읽혀 져야 한다. 만일 사용자가 핵심부를 적재하는데 다른 수단(실례로 기동디스크나 혹은 MS-DOS에 기초한 Linux적재프로그람 LOADLIN. EXE)을 리용한다면 그 구획은 아무것이라도 된다. 이 경우에 구획은 《Linux 고유》Ox83형식으로 될것이다.

사용자의 체계가 약간의 교환공간을 요구할수 있다. 또한 사용자가 파일들에 대한 교환조작을 해야 할 경우에는 지정된 교환구획을 요구하게 된다. 이 구획은 오직 Linux 핵심부에 의하여서만 접근되기때문에 PC BIOS가 없다고 하여도 Linux핵심부가 불리한 상태에 처하지 않기때문에 교환구획은 아무데나 놓여도 된다.

3.2. 교환공간배정

일반적으로 사용자가 좋은 방법으로 알려 진 하나의 교환구획을 리용하려고 결심하

면 교환구획의 크기를 평가하기 위하여 이 안내내용을 참고해도 된다.

Linux에서 RAM과 교환공간은 합쳐 져 있다(물론 모든 UNIX계에서는 아니다.). 실례로 만일 사용자가 8MB의 RAM과 12MB의 교환공간을 가지고 있으면 전체적으로 20MB의 가상기억을 가져야 한다. 그러므로 4MB의 RAM에 대하여 적어도 12MB의 교환공간을 생각할수 있으며 8MB의 RAM에 대하여서는 8MB의 공간을 생각할수 있다.

Linux에서 단일교환구획은 128MB보다 클수 없다. 즉 구획은 128MB보다 클수 있지만 초과공간은 리용될수 없다. 만일 128MB보다 큰 교환공간을 만들려면 다중교환구획을 생성해야 한다.

교환공간크기를 정할 때 너무 크게 교환공간을 정하면 전혀 쓸모가 없다는데 대하여 생각해야 한다. 매개 프로쎄스는 《작업모임》을 가지는데 이것은 제일 처음 참조하게 될 기억안페지의 모임이다. Linux는 이 기억접근을 예측하여(제일 최근에 리용한 페지들이 다시 제일 먼저 리용된다고 가정하여) 가능하면 RAM에 이 페지들을 보존하려고 한다.

만일 프로그람이 파괴된 《참조위치》를 찾는다면 이 가정은 맞는것으로 될것이며 예측알고리듬이 동작하게 된다. 주기억안의 작업모임을 유지하는 조작은 주기억이 충분할 때에만 동작가능하게 된다. 콤퓨터안에 실행중에 있는 프로쎄스가 너무 많으면 핵심부는 제일 처음에 다시 참조해야 할 디스크상에 페지들을 배치하도록 요구한다(다른 작업모임으로부터 페지를 내보내게 하고 참조된 페지안에 그 페지를 넣는다.). 보통 이 조작은 페지화조작이 극도로 증가되게 하는 결과를 발생하는데 이때 성능이 크게 떨어 진다. 이 상태에 있는 콤퓨터를 《헛수고》상태에 있다고 말한다.

《헛수고》상태에 있는 콤퓨터에서 프로쎄스들은 본질적으로 RAM으로부터가 아니라 디스크로부터 실행되게 된다. 이때의 성능은 기억접근속도와 디스크접근속도의 비에 의하여 대략적으로 떨어 지리라고 예측할수 있다.

한때 PDP와 Vax에서 리용한 변경된 규칙에 의하면 프로그람의 작업모임의 크기는 가상크기의 25%였다. 따라서 대체적으로 RAM크기의 3배보다 더 큰 교환공간을 준비하는것은 의의가 없다. 이것이 바로 "thumb"의 규칙이라는것을 상기해 둔다.

프로그람들이 매우 크거나 혹은 매우 작은 작업모임을 가지도록 동작순서를 쉽게 만들수 있다. 실례로 아주 우연적인 양상으로 접근될수 있는 큰 자료모임을 가지는 모의프로그람은 자료토막안에서 뚜렷한 참조위치를 거의 가지지 않으며 따라서 그것의 작업모임은 커지게 된다.

바꾸어 말하면 동시에 열린 많은 JPEG들을 포함한 XV는 전부는 아니지만 그림문자(icon)형식으로 되여 있으며 매우 큰 자료토막을 가지고 있다. 하지만 화상변화는 모두 하나의 단일화상우에서 실현되며 XV가 차지한 대부분의 기억은 다칠수 없다. 동시에 한개 창문만 변경시킬수 있는 여러개의 편집기창문을 가진 편집기들에서는 사실상 이것이 동일하다. 이 프로그람들은 제대로만 설계되면 고도의 위치참조성을 가지며 그 프로그람의 큰 부분은 심한 성능상 충돌이 없이 교환동작을 진행할수 있다.

누구나 명령자체가 낡았다는데로부터 25%의 수가 다중문서를 편집하는 현대적인

GUI프로그람들에서는 더이상 적합하지 않다는것을 알고 있으나 이 수자들을 검증해 볼수 있는 새로운 자료들은 물론 없다.

지금까지 16MB에 의한 배치구성에 대해서는 최소배치구성을 요구하는 교환이 전혀 없었고 48MB보다 큰 교환은 대체로 리용하지 않았다. 요구되는 기억의 정확한 크기는 콤퓨터에서 실행되는 응용프로그람에 의존한다.

3.3. 교환공간의 배치위치

기계는 뜨고 전자공학적요소는 빠르다. 현대적하드디스크들은 여러가지 자두를 가지고 있다. 같은 자리길우에서 자두들의 절환은 그 동작이 순수 전자공학적작용에 기초하므로 빠르다. 그러나 자리길사이의 절환은 기계적현상을 포함하기때문에 뜨다.

그러므로 자두가 많은 디스크나 혹은 더 적은 디스크를 가지고 있거나 두개가 서로 다른 파라메터를 가지고 있다고 할 때 많은 자두를 가지고 있는 디스크의 속도가 더 빠르다. 교환장치를 분할하고 두개의 디스크에 배치하여도 속도는 빨라 진다.

변경된 디스크들은 모든 자리길우에 동일한 쎅터수를 가지고 있다. 이러한 디스크들에서 디스크의 자두가 임의의 자리길로부터 교환대역쪽으로 움직일것이라고 가정하면 디스크의 중간에 자두를 놓는것이 제일 빠르게 될것이다.

보다 새로운 디스크들은 ZBR(기록비트대역)를 사용한다. 이 디스크들은 바깥쪽자리길에 더 많은 쎅터들을 가지고 있다. 고정된 수의 rpm에 의하여 이 방식은 안쪽의 자리길보다 바깥쪽자리길에서 더 높은 성능을 발휘한다. 그러므로 빠른 자리길에 교환기억을 놓는것이 좋다.

물론 디스크자두는 제멋대로 움직이지는 않는다. 만일 사용자가 불변동작홈, 구획 그리고 대다수 리용되지 않는 압축구획들사이에 중간크기의 교환공간을 가지고 있으면 교환장치가 보다 짧은 자두의 움직임에 대해서도 홈구획의 중간에 있을 때 상태가 더 좋아진다.

요약 다른 일을 하는 동작중이 아닌 여러개의 자두를 가진 고속디스크에 사용자의 교환공간을 배치한다. 만일 사용자가 다중디스크를 가지고 있다면 교환공간을 분할하고 그것을 모든 디스크에 혹은 다른 조종장치에 나누어 준다. 상태가 좋아 지면 더 많은 RAM을 준비할수 있다.

3.4. 파일체계와 단편화

디스크공간은 조작체계에 의하여 블로크나 블로크들의 쪼각을 단위로 관리된다. ext2 파일체계에서 쪼각들과 블로크들은 같은 크기를 가져야 하며 따라서 우리의 론의문제를 블로크에 대한 문제로 제한할수 있다.

파일은 임의의 크기를 가질수 있다. 또한 파일은 블로크경계의 끝과 일치되지 않는다. 그러므로 매개 파일에서 파일의 마지막블로크부분은 랑비된다. 파일크기가 임의라고 가정하면 디스크상에서 매개 파일에 대하여 대략적으로 블로크의 절반이 랑비된다. 타넨바움(Tanenbaum)은 이 내용을 자기의 저서 《조작체계》에서 《내부단편화》라고 불렀다.

사용자는 디스크에 배정된 색인마디의 수에 의하여 디스크상의 파일수를 추측할수 있다. 디스크상에서 이 내용은 다음과 같이 서술된다.

df −i

Filesystem	Inodes	Iused	Ifree	%Iused	Mounted on
/dev/hda3	64256	12234	52022	19%	/
/dev/hda5	96000	43058	52942	45%	/var

보는바와 같이 /우에는 약 12000개의 파일이 있으며 /var에는 44000개의 파일이 있다. 만일 블로크크기를 4KB로 선택했다면 이 공간의 4배를 잃을수도 있다. 자료전송은 큰 자료의 련속덩어리에서 더 빠르다. 그것은 ext2이 파일에 대하여 8개의 련속된 블로크단위로 공간을 선행배정하기때문이다.

리용되지 않는 선행배정은 파일이 닫길 때 개방되며 따라서 랑비되는 공간은 없다. 파일안에 블로크들을 불련속적으로 배치하는것은 성능상 견지에서 좋은것이 못된다. 왜냐하면 파일들은 흔히 순차적방식으로 접근되기때문이다.

조작체계들에서는 디스크접근을 분할하도록 요구하며 디스크가 자두를 움직이도록 요구한다. 이것을 《외부단편화》 혹은 단순히 《단편화》라고 부르며 이것은 DOS파일체계에서 공통적문제이다.

ext2파일체계는 외부단편화를 극복하기 위한 몇가지 방책을 가지고 있다. 표준적으로 ext2에서 단편화는 큰 문제가 아니며 지어 USENET 새 소식 spool과 같은 구획에도 중요하게 리용되지 않는다.

ext2파일체계의 비단편화를 위한 도구가 있기는 하지만 누구도 그것을 리용하지 않으며 현재까지도 ext2의 현행판본에서 리용하지 않는다.

MS−DOS파일체계가 디스크공간의 관리를 치료적으로 진행한다는데 대해서는 잘 알려 져 있다. MS−DOS에 의하여 리용된 아주 불량한 캐쉬와 관련하여 파일단편화의 효과는 아주 주목할만하다. DOS사용자들은 몇주일에 한번씩 디스크들의 쪼각을 모아 두는데 습관되였으며 일부 사람들은 쪼각모으기를 전문화하는 의식적인 도구까지 개발하였다. Linux와 ext2에서는 이러한 습관적행동을 취할 필요가 없다.

MS−DOS파일체계는 또한 내부적인 단편화로 인하여 많은 량의 디스크공간을 루실하는것으로 알려 져 있다. 256MB보다 큰 구획에서 DOS블로크크기는 커지게 되며 따라서 더이상 쓸모 없게 된다(이 부족점은 FAT32에 의한 일부 확장체계에서는 수정되였다.). ext2는 사용자로 하여금 0.5TB나 그이상의 아주 큰 파일체계(1TB는 1024GB)들을 제외하고는 큰 파일체계들의 블로크들을 선택할수 있게 한다. 0.5TB이상 규모의 큰 파일체계에서는 작은 블로크크기가 효과성을 가지지 못한다. DOS와 다른 체계들에서는 큰 디스크들을 블로크크기가 작은 다중구획들로 쪼갤 필요가 없다. 가능하면 블로크크기를 기정값으로 지정된 1KB를 사용하면 된다. 사용자가 일부 구획에 대하여 2KB의 크기를 가진 블로크를 실험해 보려 하지만 오유가 생길수 있다고 예측할수 있다. 대부분의 사람들은 기정값을 쓴다.

3.5. 구획평가기준으로서의 파일수명과 여벌복사주기

ext2에 의하여 구획을 설정하려고 하면 서로 다른 파일수명으로부터 외부단편화를 피하기 위한 여벌복사방법으로 관리하여야 한다. 파일은 수명을 가지고 있다. 파일은 일단 생성된후에 일정한 시간동안 체계에 남아 있으며 후에 제거된다. 파일수명은 체계전체에 걸쳐 크게 달라 지며 특히 파일의 경로이름에 부분적으로 의존한다. 실례로 /bin, /sbin, /usr/sbin, /usr/bin과 그와 류사한 등록부들에 있는 파일이름은 매우 긴 파일수명(몇달 혹은 그이상)을 더 가지는것처럼 보인다. /home등록부에 있는 파일들은 중간정도의 수명 즉 몇주일 혹은 그이상의 수명을 가지는것 같이 보인다. /var에 있는 파일들은 보통 수명이 짧다. /var/spool/news의 파일들은 며칠이상 남아 있게 되며 /var/spool/lpd의 파일들은 그 수명이 몇분 혹은 그보다 더 적게 측정된다.

여벌복사에서 수명은 매일매일의 여벌복사량이 단일한 여벌복사의 중간정도의 용량보다 더 작으면 아주 쓸모 있다. 매일매일의 여벌복사는 완전여벌복사로 될수 있거나 혹은 증가형여벌복사로 된다.

사용자는 구획의 크기가 하나의 여벌복사의 중간값에 완전히 일치시킬수 있는 정도로 충분히 작게 보존되도록 할수 있다. 임의의 경우에 구획은 매일매일의 델타(변경된 모든 파일)가 하나의 여벌복사의 중간에 일치시킬수 있을 정도로 작아야 한다. 사용자의 여벌복사방책은 그것의 결정에 의존한다.

자료의 재생성비용은 가상적으로 임의의 정보에 대한 여벌복사비용보다 훨씬 더 높다. 성능을 보존하기 위하여 서로 다른 구획상에서 서로 다른 생명주기를 가진 파일들을 보존하는것이 아주 좋다. 새 소식구획우에서 짧은 수명을 가진 파일들을 단편화할수 있는 방법은 대단히 중요하다. 이 방법은 /구획이나 /home구획의 성능에 영향을 주지 않는다.

4. 실 례

4.1. 초학자들을 위한 권고안

우에서 언급된것처럼 공통모형은 /, /home, /var구획을 생성한다. 이 모형을 설치하고 유지하는것은 단순하며 서로 다른 수명으로부터 불리한 효과들을 충분히 구별할수 있다. 또한 이 모형은 여벌복사모형과도 잘 일치한다. USENET 새 소식spool을 여벌복사하는데는 근심거리가 거의 없으며 /var의 일부 파일들만 여벌복사할 가치가 있다. 다른 말로 말하여 /는 이따금 변하며 요구에 따라 여벌복사할수 있으며 대다수의 현대적매체에 완전여벌복사로 일치시킬수 있을만큼 충분히 작다(설치된 프로그람의 크기에 따라 250~500MB를 예견한다.). /home는 쓸모 있는 사용자자료를 포함하며 매일 여벌복사하여야 한다. 일부 설치는 매우 큰 /home을 가지며 증가형여벌복사를 리용해야 한다.

일부 체계들은 /tmp를 개별적디스크에 배치하고 있으며 다른 체계들은 동일한 효과를 얻기 위하여 /var/tmp에 그것을 기호련결로 련결한다(이 조작은 단일사용자방식에서 효과적이지만 이 방식에서는 /var를 리용할수 없다. 체계는 사용자가 /var를 생성하거나 /var를 수동적으로 올려태우기할 때까지 혹은 RAM디스크에 배치할 때까지 이것을 가질수 없다는데 대하여 강조한다.).

이 모형은 갱신뿐아니라 재설치에서도 아주 편리하다. 사용자의 배치구성파일(혹은 /etc전체)을 /home등록부에 기억시키고 /등록부를 제거하며 /home상에 기억된 등록부로부터 변경된 배치구성을 재설정하고 선행불러내기한다.

5. 실현방법

변경된 ISA bus 386/40을 홈용LAN을 위한 소형 x-less봉사기로 돌린 경험을 소개한다. 386을 선택하고 거기에 16MB RAM을 설치하여 쉽게 얻을수 있는 제일 작은 디스크인(800MB) 값 눅은 EIDE디스크와 ethernet카드를 추가한다. Linux가 설치되고 현시장치도 포함되여 있었기때문에 Herclules도 추가하였다. 다음에 국부NFS, SMB, HTTP, LPD/LPR와 NNTP봉사기뿐만아니라 우편경로조종기와 POP3봉사기를 결합하였다. 보충적인 ISDN카드에 의하여 콤퓨터는 TCP/IP경로조종기로도 되고 방화벽으로도 되였다. 이 기계상에서 대부분의 디스크공간은 /var등록부, /var/spool/mail, /var/spool/news 그리고 /var/httpd/
htm1로 되였다. 다음 /var를 개별디스크에 놓고 이것을 한개 등록부로 크게 만들었다. 이 기계상에 더이상 사용자들이 없기때문에 홈구획은 만들지 않고 NFS를 거쳐 다른 워크스테이션으로부터 /home을 올려태우기하였다. X와 몇개의 국부적인 설치나 편의프로그람없이 Linux로써 250MB구획으로 잘 진행되였다. 기계가 16MB의 RAM을 가지고 있었지만 많은 봉사기들을 운영할수 있었으며 16MB이면 적당하였고 32MB이면 충분하였다. 여기서 얻은 내용은 다음과 같다.

장 치	올려태우기점	크 기
/dev/hda1	/ dos_c	25MB
/dev/hda2	− (Swapspace)	32MB
/dev/hda3	/	250MB
/dev/hda4	− (Extended container)	500MB
/dev/hda5	/ var	500MB
homeserver : /home /home		1.6GB

홈봉사기에서 테프를 리용하는 망을 거쳐 이 기계와 접속된다. 이 기계상의 모든것이 CD−ROM으로부터 설치되였기때문에 /etc로부터 일부 배치구성파일들과 /root/source로부터 주문화되고 국부적으로 설치된 *. Tgz를 기억시킨다. 그리고 /var/spool/mail뿐아니라 /var/httpd/htm1도 기억시킨다. 이 파일들을 매일밤 홈봉사기의 /home/backmeup등록부에 복사하는데 정규홈봉사기여별복사가 그것들을 선택한다.

색 인

ㅁ

ㅂ

ㅅ

ㅇ